Adaptive Indexing for Complex Data

Konstantinos Lampropoulos

Ph.D. Dissertation

— ♦ —

Ioannina, April 2025

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Adaptive Indexing for Complex Data

A Dissertation

submitted to the designated
by the Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

Konstantinos Lampropoulos

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Ioannina
School of Engineering
Ioannina 2025

Advisory Committee:

- **Nikos Mamoulis**, Professor, Department of Computer Science and Engineering, University of Ioannina (Advisor)
- **Panagiotis Vassiliadis**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Panagiotis Bouros**, Assistant Professor, Institute of Computer Science, Johannes Gutenberg University Mainz (JGU)

Examining Committee:

- **Nikos Mamoulis**, Professor, Department of Computer Science and Engineering, University of Ioannina (Advisor)
- **Panagiotis Vassiliadis**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Panagiotis Bouros**, Assistant Professor, Institute of Computer Science, Johannes Gutenberg University Mainz (JGU)
- **Panagiotis Tsaparas**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
- **Apostolos Zarras**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Panagiotis Karras**, Professor, Department of Computer Science, University of Copenhagen
- **George Papastefanatos**, Principal Researcher, Information Management Systems Institute (IMSI) of the Athena Research Center.

Dedication

To my beloved family

Acknowledgements

First and foremost, I would like to express my heartfelt gratitude to my supervisor, Nikos Mamoulis, for his unwavering support, guidance, and patience, as well as for giving me the opportunity to work alongside him. I would also like to sincerely thank Professor Panagiotis Karras for his valuable guidance and for sharing his expertise. Additionally, I am grateful to my colleagues at both Ioannina and Aarhus for their support and for contributing to a positive, collaborative work environment. Last but not least, I want to express my deep appreciation to my family for their constant encouragement and for motivating me to continue pushing toward my goals.

Konstantinos Lampropoulos

April 2025

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Konstantinos Lampropoulos, Ph.D., Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, 2025.
Adaptive Indexing for Complex Data.
Advisor: Nikos Mamoulis, Professor.

As data continues to grow in both volume and complexity, especially in the context of multidimensional datasets, traditional indexing methods often fail to offer efficient solutions for large-scale data exploration. Constructing an index upfront can be costly and inefficient, particularly when query volumes are low or have unpredictable patterns. Adaptive indexing addresses this challenge by dynamically building and optimizing an index incrementally, following the query workload. This approach ensures that the indexing structure evolves to meet the specific needs of the queries being executed, reducing the cumulative cost of index construction and usage. It proves particularly advantageous in environments where query workloads are small or skewed. By building an index only on relevant data, adaptive indexing offers an efficient, flexible solution for accelerating exploratory search operations without the high cost of constructing and maintaining a pre-built index. This is especially beneficial for data analysis tasks, where the goal is to query large, multidimensional datasets stored in main memory efficiently.

Adaptive indexing has shown success for single-attribute or simpler data models; however, it encounters challenges when applied to complex spatial data objects and multidimensional range queries. Existing methods for multidimensional adaptive indexing partition space into orthotopes (hyperrectangular units), but this approach is highly ineffective in high-dimensional spaces. To address this limitation, we propose an alternative method for adaptive high-dimensional indexing that partitions the space around query centers into units defined by hyperspheres, leveraging previously computed distances, with the query centers serving as vantage points.

Several adaptive indexing techniques have been developed for multidimensional range queries, each with its own strengths and weaknesses. There is a lack of comparative studies that evaluates these methods under diverse conditions, including different data types, distributions, sizes, and workload patterns. To fill this gap, we have developed a comprehensive benchmark to rigorously evaluate the performance, strengths, and weaknesses of existing multidimensional adaptive indexing methods across various scenarios, providing valuable insights that complement previous research. Additionally, we propose technical extensions that enhance the efficiency of existing methods.

Finally, we note that existing spatial adaptive indexing methods are generally designed for static data, available in a one-off manner. To date, no spatial adaptive indexing method can accommodate interleaved data updates during data exploration. We propose an update mechanism for adaptive in-memory indices for multidimensional objects, enabling the index to absorb data insertions as they arrive while maintaining up-to-date accuracy. Our design integrates insertions into the structure progressively, allowing them to gradually move down the hierarchy as they accumulate, while reorganizing the underlying data array by moving and splitting partitions.

In summary, this dissertation provides a comprehensive exploration of adaptive indexing techniques for multidimensional data, addressing key challenges in efficiently handling large-scale data exploration and complex query workloads. It introduces a novel approach to high-dimensional adaptive indexing by leveraging query centers as vantage points, overcoming the limitations of traditional partitioning methods. Through a proposed benchmark, the dissertation systematically evaluates existing multidimensional adaptive indexing techniques across various data types, distributions, and query patterns, offering valuable insights for optimizing indexing performance. Furthermore, it presents a unique update mechanism that enables dynamic adaptation to real-time data insertions and deletions, ensuring the index remains up to date during data exploration. These contributions significantly advance the field of adaptive indexing, providing practical solutions for managing and querying complex, multidimensional data in dynamic environments.

Εκτεταμενη Περιληψη

Κωνσταντίνος Λαμπρόπουλος, Δ.Δ., Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, 2025.
Adaptive Indexing for Complex Data.
Επιβλέπων: Νίκος Μαμουλής, Καθηγητής.

Καθώς τα δεδομένα συνεχίζουν να μεγαλώνουν τόσο σε όγκο όσο και σε πολυπλοκότητα, ειδικά όταν τα σύνολα δεδομένων είναι πολυδιάστατα, τα παραδοσιακά ευρετήρια συχνά αποτυγχάνουν να προσφέρουν αποτελεσματικές λύσεις για εξερεύνηση δεδομένων μεγάλης κλίμακας. Η κατασκευή ενός ευρετηρίου μπορεί να είναι δαπανηρή και αναποτελεσματική, ιδιαίτερα όταν ο όγκος των ερωτημάτων είναι μικρός ή όταν τα ερωτήματα δεν είναι ομοιόμορφα κατανεμημένα στο χώρο. Η προσαρμοστική ευρετηρίαση (adaptive indexing) αντιμετωπίζει αυτή την πρόκληση δημιουργώντας και επεκτείνωντας ένα ευρετήριο σταδιακά, προσαρμοσμένο στα ερωτήματα. Αυτή η μέθοδος εξασφαλίζει ότι η δομή του ευρετηρίου εξελίσσεται για να καλύψει τις ανάγκες των ερωτημάτων που αποτιμούνται, μειώνοντας το συνολικό κόστος της κατασκευής και συντήρησης του. Αποδεικνύεται ιδιαίτερα αποδοτική σε καταστάσεις όπου ο φόρτος των ερωτημάτων είναι μικρός ή ιδιόμορφος. Με τη δημιουργία ενός ευρετηρίου μόνο σε σχετικά δεδομένα, η προσαρμοστική ευρετηρίαση προσφέρει μια αποτελεσματική, ευέλικτη λύση ιδανική για εργασίες διερευνητικής αναζήτησης αποφεύγοντας το υψηλό κόστος κατασκευής και διατήρησης ενός προκατασκευασμένου ευρετηρίου. Αυτό είναι ιδιαίτερα ωφέλιμο στην ανάλυση δεδομένων, όπου σκοπός είναι η αναζήτηση σε μεγάλα, πολυδιάστατα σύνολα δεδομένων που είναι αποθηκευμένα στην κύρια μνήμη.

Η προσαρμοστική αναζήτηση έχει αποδειχθεί πολύ αποδοτική για απλά δεδομένα ή μιας διάστασης, ωστόσο αντιμετωπίζει προβλήματα όταν εφαρμόζεται σε πολύπλοκα χωρικά δεδομένα και πολυδιάστατα ερωτήματα εύρους. Οι υπάρχουσες μέθοδοι για προσαρμοστική ευρετηρίαση πολυδιάστατων δεδομένων, χωρίζουν

το χώρο σε (υπερ)ορθογώνια, κάτι που είναι εξαιρετικά αναποτελεσματικό σε πολυδιάστατους χώρους. Για να αντιμετωπίσουμε αυτό το πρόβλημα, προτείνουμε μια μέθοδο για την προσαρμοστική ευρετηρίαση δεδομένων υψηλής διάστασης, η οποία χωρίζει το χώρο γύρω από τα ερωτήματα χρησιμοποιώντας υπερ-σφαιρικές δομές και αξιοποιεί προηγουμένως υπολογισμένες αποστάσεις.

Αρκετές τεχνικές προσαρμοστικής ευρετηρίασης έχουν αναπτυχθεί για πολυδιάστατα ερωτήματα εύρους, καθεμία με τα δικά της πλεονεκτήματα και μειονεκτήματα. Ωστόσο, δεν υπάρχει μια συγκριτική μελέτη που να αξιολογεί αυτές τις μεθόδους υπο διαφορετικές συνθήκες, συμπεριλαμβανομένων δεδομένων διαφορετικού τύπου, κατανομής, μεγέθους και διαφορετικών ερωτημάτων. Για να καλύψουμε αυτό το κενό, προτείνουμε μια ολοκληρωμένη μελέτη για την αξιολόγηση της απόδοσης, των δυνατοτήτων και των αδυναμιών των υφιστάμενων προσαρμοστικών ευρετηρίων για πολυδιάστατα δεδομένα σε ποικίλα σενάρια, παρέχοντας πολύτιμα ευρήματα που συμπληρώνουν την προτερη έρευνα. Επιπλέον, προτείνουμε τεχνικές επεκτάσεις που βελτιώνουν την αποτελεσματικότητα των υφιστάμενων μεθόδων.

Τέλος, παρατηρούμε ότι τα υπάρχοντα προσαρμοστικά ευρετήρια για χωρικά δεδομένα είναι σχεδιασμένα για στατικά δεδομένα. Μέχρι σήμερα, κανένα τέτοιο ευρετήριο δεν μπορεί να διαχειριστεί ενημερώσεις. Προτείνουμε ένα μηχανισμό που επιτρέπει σε προσαρμοστικά ευρετήρια κύριας μνήμης για πολυδιάστατα δεδομένα να δέχεται εισαγωγή δεδομένων διατηρώντας την ακρίβεια και αποτελεσματικότητά του. Η σχεδίαση μας ενσωματώνει την εισαγωγή δεδομένων στη δομή προοδευτικά. Παράλληλα η δομή αναδιοργανώνεται μετακινώντας και διαμερίζοντας τα δεδομένα.

Συνοψίζοντας, αυτή η διατριβή παρέχει μια ολοκληρωμένη εξερεύνηση της προσαρμοστικής ευρετηρίασης για πολυδιάστατα δεδομένα, αντιμετωπίζοντας σημαντικές προκλήσεις για τον αποτελεσματικό χειρισμό δεδομένων μεγάλης κλίμακας και σύνθετων ερωτημάτων. Επιπλέον, εισάγει μια νέα προσέγγιση για την ευρετηρίαση δεδομένων υψηλής διάστασης χρησιμοποιώντας τα ερωτήματα ως σημεία αναφοράς, ξεπερνώντας τους περιορισμούς των παραδοσιακών ευρετηρίων. Στη συνέχεια, η διατριβή αξιολογεί συστηματικά τα υπάρχοντα προσαρμοστικά ευρετήρια για πολυδιάστατα δεδομένα σε ένα ευρύ φάσμα δεδομένων και ερωτημάτων, προσφέροντας σηματνικά ευρήματα για την βελτίωση της απόδοσης τους. Τέλος, παρουσιάζει ένα μηχανισμό ενημερώσεων που επιτρέπει τη δυναμική προσαρμογή του ευρετηρίου σε εισαγωγές και διαγραφές δεδομένων σε πραγματικό χρόνο, δια-

σφαλίζοντας την αξιοπιστία και αποδοτικότητα του. Αυτές οι συνεισφορές, προάγουν τον τομέα της προσαρμοστικής ευρετηρίασης, παρέχοντας πρακτικές λύσεις στη διαχείριση πολυδιάστατων δεδομένων και αναζήτηση σύνθετων ερωτημάτων σε δυναμικά περιβάλλοντα.

Chapter 1

Introduction

Data management is an essential component of modern computing, focusing on the processes of collecting, storing, organizing, retrieving, and maintaining data in an efficient and effective manner. As the volume and complexity of data increase across various industries, ranging from healthcare to finance to social media, traditional data management systems that rely on static and predefined structures are often insufficient. These systems are typically designed for specific use cases and struggle to meet the diverse and growing demands of modern data. Efficient data management is essential for storing large datasets in a way that ensures fast and easy access, while also maintaining consistency and minimizing redundancy. A key challenge in data management lies in selecting the right methods for accessing and organizing data to optimize both storage and retrieval times, particularly as queries become more complex and diverse. As datasets grow larger and more varied, especially with the rise of big data, traditional methods may no longer be sufficient. New approaches are needed to ensure that data management remains responsive to ever-changing user queries and evolving business requirements.

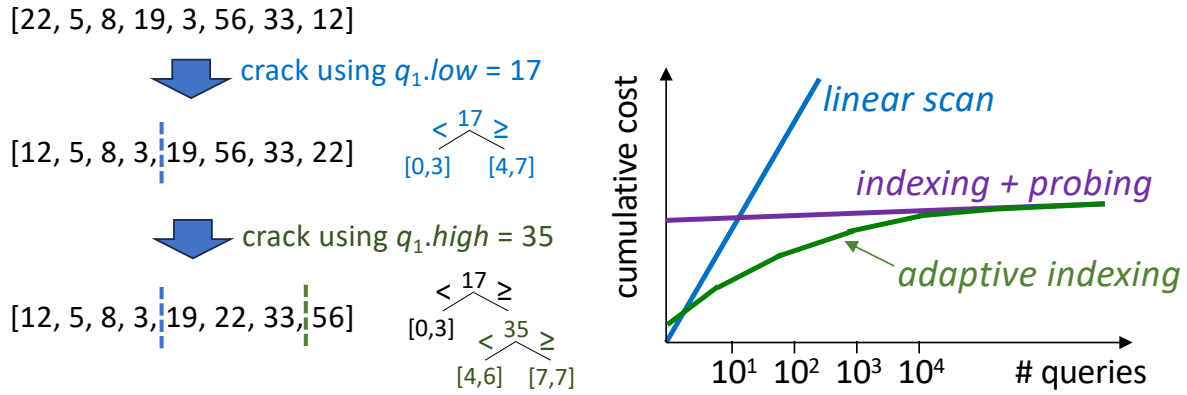
One such innovative approach to addressing the challenges of modern data management is adaptive indexing. Adaptive indexing is a dynamic technique that enhances the performance of databases by automatically adjusting the structure and creation of indices based on query patterns and data access behaviors. Unlike traditional indexing methods, which require manual configuration and can become inefficient as data grows, adaptive indexing allows the system to continuously adapt to the queries being executed and optimize the indexing strategy accordingly. This self-tuning mechanism helps improve query performance, reduce unnecessary index overhead, and ensure that data retrieval is fast and efficient, even as datasets become larger and more complex. As a result, adaptive indexing is becoming an essential tool for managing data in environments where the volume and diversity of queries are constantly evolving.

Data accessing methods are the backbone of how systems interact with stored data, and choosing the appropriate method has a significant impact on performance. Various methods of data access exist, each suitable for different types of workloads. For instance, methods like linear scanning, hashing, and tree-based structures are commonly used for data retrieval. Linear scanning involves searching through data sequentially, which works well for small datasets but can be inefficient as the size of the dataset grows. Hashing, while providing fast access for certain types of data, lacks flexibility when it comes to more complex queries. Tree-based structures, such as B-trees or binary search trees, allow for efficient searching by organizing data hierarchically. These structures help minimize the number of comparisons required to find specific data elements, thus significantly improving performance.

Indexing transforms data retrieval from potentially slow, linear searches into highly optimized queries that can return results almost instantaneously, even in large-scale datasets. However, as data grows in complexity and the range of possible queries expands, the upfront construction cost of pre-built indices does not pay off, especially when queries are few or skewed. This is where more sophisticated indexing approaches, like adaptive indexing, come into play, as they allow systems to evolve their structure in real-time based on actual usage patterns, thereby improving data access efficiency and minimizing the cumulative cost of index construction and query evaluation.

1.1 Adaptive Indexing

Adaptive indexing has been extensively studied in the database community since 2007 [1, 2]. Consider an unorganized column C of a relation, e.g., in a column store [3, 4]. Adaptive indexing constructs an in-memory index for C progressively and in response to range (or equality) queries along a single dimension or attribute [5]. Assume the first range query q_1 seeks records r such that $q_1.low \leq r.C < q_1.high$. An adaptive index 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 index the trichotomy. Figure 1.1a shows an example with $q_1.low = 17$ and $q_1.high = 35$. The array is first *cracked* via one iteration of quicksort [6] using 17 as pivot. Elements are swapped accordingly to create a binary tree rooted at 17, having as leaves array position ranges $[0, 3]$ and $[4, 7]$. Then, the tree is searched for 35, cracking the subarray on its right leaf on 35, to gather all query results between the cracks, i.e., at positions 4 to 6. 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 1.1: Adaptive indexing

Figure 1.1b 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 and probing it for each query.

Adaptive indexing facilitates the exploration of large short-lived datasets that become available in batches with not many, unpredictably distributed queries [7, 8, 9, 10]. 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.

1.2 Adaptive indexing in high-dimensional metric spaces

Let O be a set of objects in a (high-dimensional) metric space. Given a query object q, a distance bound ϵ, and a distance metric $d()$, a *range similarity query* seeks the objects $o \in O$ for which $d(q, o) \leq \epsilon$. Similarly, given a positive integer k, a *k-nearest-neighbor (kNN) similarity query* seeks k objects $o \in O$ having smaller $d(q, o)$ than all other objects in O. Range and kNN similarity queries are routinely used in similarity-based search and data mining tasks (e.g., clustering [11], NN classification and outlier detection [12]) for application domains including computer vision [13], information retrieval [14], kNN search in spatial networks [15], and recommender systems [16].

We consider applications where data in a metric space are short-lived and a relatively small number of queries is expected before the data become obsolete. For example, satellite images that depict weather phenomena or other transient information are periodically received and automatically converted to feature vectors appropriate for similarity computations. Data scientists perform similarity search against the image collection to detect phenomena. Such images become obsolete when the next batch arrives, hence the number of queries applied on one batch is not expected to be large.

In such environments, building an index prior to query processing for each batch of data is costly and may thus not be worthwhile. Instead, one may evaluate each query directly on the raw feature vectors by linear scan.

Given the modern size of memories and the fact that we target applications where the data are short-lived, hence not voluminous, we assume that the data are stored in memory, like the majority of previous work in adaptive indexing [1, 17, 18]. For example, a collection O of objects in a D-dimensional vector space can be stored in a data array as a sequence of *feature vectors* $\langle o_{id}, o_1, o_2, o_3, \ldots, o_D \rangle$, where o_{id} is the identifier of object $o \in O$ and o_i is the value of the object in the i-th dimension (feature). Let (q_1, ϵ_1) be the first (range) query. While linearly scanning the data array to derive query results, we conduct object *swaps* to *crack* the array in two pieces: one piece containing all objects that are query results and another all remaining objects. At the same time, we initialize an *adaptive vantage* tree (AV-tree) with (q_1, ϵ_1) as root. As new queries arrive, we compare them to past queries using the AV-tree, and search only the parts of the array that may contain query results. Guided by the triangle inequality, we avoid accessing irrelevant fragments of the array, while introducing cracks and tree nodes corresponding to new queries. To prevent the tree from becoming excessively large, we abide by a threshold θ for cracked pieces; if a piece has fewer elements than θ, then it is *fixed* and not further cracked. Further, we find that cracking based on the *median distance* of all data points in a piece rather than the current query bound ϵ_i results in a much better index. In addition, in *fixed* pieces we cache previously computed distances and exploit the sort order to achieve an *early termination* of comparisons.

Contributions. We propose AV-tree, a method for adaptive high-dimensional indexing that exploits previously computed distances, using query centers as vantage points. Additionally, it is the first multidimensional adaptive index that supports kNN queries. The AV-tree is not only applicable in vector spaces where, for example, an L_p-norm distance (e.g., Euclidean distance) is used, but also in general metric spaces; e.g., for indexing a collection O of strings to support similarity search based on edit distance.

The contributions of the first chapter can be summarized as follows:

- We investigate, for the first time, the problem of building a *distance-based* multidimensional adaptive index.

- We define the AV-tree, an index that efficiently adapts to the query workload, forming a *unified* solution for both distance range queries and kNN queries.
- We provide several enhancements on the AV-tree.
- We conduct an extensive experimental study, showing that the AV-tree behaves as an ideal adaptive index should.

1.3 Benchmarking Adaptive Multidimensional Indices

In-memory adaptive indexing of multidimensional data has been an area of interest for several years [19, 20, 18, 21, 22, 23, 24]. 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 indices. While 1D adaptive indices have reached a maturity level already 10 years ago and have been extensively evaluated [25, 26], multidimensional adaptive indexing is an active research area with several recent developments [18, 21, 22] that have yet to be evaluated within the same framework. Besides, ideas developed for one index (e.g., as in [21]) have not been tested on other structures (e.g., those in [18]) and composite solutions for the 1D case [25] 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 [27] compared QUASII [19] and AKD [28], the only available methods at the time. Variations of these methods were also evaluated in [18]. However, the studies in [27, 18] only target point data and do not consider hybrid schemes that combine multidimensional partitioning with cracking, as in [25] for 1D data. A later study [21] reevaluated state-of-the-art methods, yet focuses on low-dimensional spatial data. Besides, an adaptive index for metric spaces [22] was not included in those studies. We fill these gaps via the following contributions:

- We comprehensively evaluate existing multidimensional adaptive indices using

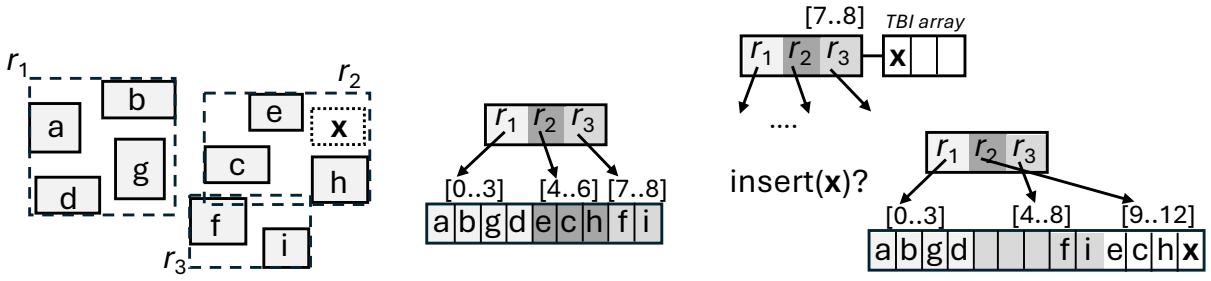
real and synthetic datasets and workloads of diverse (i) data types (point or box), (ii) dimensionality, (iii) data distribution, (iv) size distribution (for boxes), (v) query distribution, and (vi) query order. Thereby, we determine the superiority of certain techniques within various areas of the problem space.

- We apply enhancements proposed for AIR [29] to AKD [18] to craft the Advanced AKD (AAKD).
- We implement the concept of Course Granular Index (CGI) [25] 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 [22].
- 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.

1.4 Updating an Adaptive Spatial Index

Existing spatial adaptive indexing methods cater to static data made available in an one-off manner. To date, no spatial adaptive indexing method can ingest data updates interleaved with data exploration. An extension of database cracking [30] provisionally stores newly inserted data in a log and triggers their insertion in the data array only once they become relevant to a query by a *rippling* strategy that recursively moves items from one array partition to the next, until it reaches the end of the array or a piece not relevant to the query; in the latter case, it moves some data to the log, to be reinserted in response to future queries. Deletions are also materialized once they become relevant to a query, creating empty array positions (*holes*) that are used whenever possible to accommodate insertions.

To our knowledge, no existing method handles updates intertwined with queries on an adaptive *spatial* index. The challenge is that all data are packed in a single array, so inserting a new object (e.g., object x) in a leaf would require fitting a new item in a packed subarray (e.g., subarray [4..6] pointed by r_2 in Figure 1.2b). Updates on an adaptive index have only been treated in 1D in a rudimentary manner [30]. To fit



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

Figure 1.2: Spatial adaptive indexing.

newly inserted data values relevant to a query in the partitioned 1D array, a *rippling* strategy [30] repeatedly swaps data from one partition to the next, until it reaches a partition outside the query range, whereupon it pushes some data into a temporary log to make space space for the new items. Rippling is imposed by the total order that the cracked array pieces should follow. However, in multidimensional indexing, there is no requirement that the partitions follow a total order.

Table 1.1 positions GLIDE in relation to alternative ways to handle spatial data subject to an unpredictable workload of queries and updates. Classic methods, such as the R-tree and the Quad-tree, have an immense *startup cost* to construct the index before processing a workload. In addition, updating a fully grown index is quite expensive, yielding a high cumulative cost for construction and usage. An alternative, denoted as SAI (Spatial Adaptive Index)+Scan, incrementally constructs an adaptive spatial index on the initially unorganized array (as in previous work [21, 28, 19]), while appending insertions at the end of a separate array, keeping them unorganized. This alternative has low insertion cost, yet its query processing cost remains relatively high throughout the workload, as initial queries apply on an unformed index and later ones have to scan the unorganized inserted data. GLIDE confers the advantages expected from an adaptive index, i.e., zero startup cost and decreasing query cost, while also gradually ingesting insertions with consistently low cumulative cost.

In summary, we propose GLIDE, a versatile update management module applicable to any tree-based multidimensional index built by adapting to queries [28, 19, 29, 22]. To design GLIDE, we consider eager vs. lazy and update-driven vs. query-driven options for ingesting query-intertwined data updates into a single data array of contiguous leaf buckets; we opt for a lightweight design that lets inserted data objects gradually trickle down the index, provisionally residing at internal nodes at

Table 1.1: GLIDE vs. other ways to update a spatial index.

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

any level, as illustrated at the top of Figure 1.2(c). In case an array partition grows too large, GLIDE moves a part to the array's end, leveraging the flexibility to locate data in the multidimensional case, while also introducing *holes* (i.e., empty slots) in the array to efficiently accommodate future insertions, as illustrated at the bottom of Figure 1.2(c). Our thorough experimental analysis shows that GLIDE robustly offers superior overall performance across query-to-insertion ratios on real-world data sets.

1.5 Dissertation Outline

The rest of this dissertation is organized as follows. In Chapter 2 we review related work and present in detail the characteristics and weaknesses of existing methods.

In Chapter 3 we present the AV-tree, an adaptive index designed for similarity search in high-dimensional metric spaces that exploits previously computed distances, using query centers as vantage points. The AV-tree partitions the space around query centers into units defined by hyperspheres that naturally adapt to the data distribution.

In Chapter 4, we present a comprehensive benchmark that thoroughly evaluates the performance, strength, and limitations of existing multidimensional adaptive indexing methods across diverse scenarios contributing valuable insights that complement previous works. In addition, we suggest complementary technical extensions that enhance the efficiency of existing methods.

In Chapter 5 we introduce GLIDE, a novel method that intertwines adaptive indexing and incremental updating of a spatial-object data set. GLIDE builds a hierarchical spatial index incrementally in response to queries and also ingests updates judiciously into it. We examine several design choices and settle for a variant that combines gradual self-driven top-down insertions with query-driven indexing operations.

Finally, Chapter 6 summarizes the contributions of this dissertation and provides a discussion about future work.

Chapter 2

Background & Related Work

2.1 Spatial Indices

Spatial indices are data structures specifically designed to optimize the querying and retrieval of spatial data, making them crucial for efficiently handling large datasets in applications involving geographic, image, or multidimensional data. These indices are typically constructed ahead of time, before any query is executed, to speed up search operations. They are designed to quickly locate data points or objects within a defined spatial region, thereby reducing the computational cost of performing queries like range queries or nearest neighbor searches. Rather than scanning the entire dataset for each query, spatial indices enable targeted, efficient searches, which is especially beneficial in applications dealing with large, complex datasets, such as Geographic Information Systems (GIS), computer graphics, and spatial databases. In this section, we will explore several popular prebuilt spatial indexing structures—R-trees, Quad-trees, and k-d trees—that are commonly used for organizing and searching spatial data efficiently.

The R-tree [31] is a dynamic, height-balanced tree data structure primarily used for indexing spatial objects, such as points, lines, and polygons, in multi-dimensional spaces. Unlike traditional B-trees [32], which are designed for indexing one-dimensional data like numbers or strings, R-trees are specifically designed to index spatial data, where objects can have multiple dimensions, such as two-dimensional data (e.g., latitude and longitude) or three-dimensional data (e.g., x, y, and z coordinates). The key idea behind the R-tree is to represent spatial objects by enclosing them in the smallest bounding box that contains the entire object, known as the Minimum Bounding Rectangle (MBR).

In an R-tree, each node contains a set of entries, and each entry consists of two parts: a bounding box that encompasses a subset of spatial objects and a pointer to the next level of the tree or the actual data. The root node is at the top level and contains the broadest bounding boxes, while leaf nodes contain the actual spatial data or references to the data.

One of the main advantages of an R-tree is its ability to efficiently perform spatial queries, such as range queries (finding all objects within a specific area) and nearest neighbor searches (finding the closest object to a given point). These types of queries can be computationally expensive, especially in large datasets, but the hierarchical structure of the R-tree reduces the search space by pruning irrelevant areas of the data. This makes R-trees particularly suitable for applications that need to handle large datasets, such as Geographic Information Systems (GIS), spatial databases, and location-based services.

Additionally, R-trees are designed to be dynamic, allowing for efficient insertion, deletion, and updates of spatial objects. However, the performance of R-trees can degrade in certain scenarios due to inefficiencies in the node splitting process and excessive overlap of Minimum Bounding Rectangles (MBRs), which can lead to poor query performance. Specifically, suboptimal splits can result in high overlap between MBRs, causing more nodes to be visited during range queries and nearest neighbor searches. Furthermore, inefficient space utilization within the tree can lead to slower searches and larger trees. To address these issues, variants of the R-tree, such as the R*-tree [33], have been developed. The R*-tree introduces improved strategies for node splitting, minimizing MBR overlap, and reinsertion of data to optimize tree balance and performance, especially in large and dynamic datasets.

A Quad-tree [34] is a tree data structure commonly used for partitioning two-

dimensional spaces by recursively subdividing the space into four quadrants or regions. It is particularly useful for spatial indexing and efficient querying of data points in 2D spaces, such as geographical maps, images, or any data that can be represented by spatial coordinates. In a quad-tree, each node represents a region of space, and the tree recursively subdivides the region into four smaller, equally sized quadrants. These quadrants are stored as child nodes, and this process continues until a certain condition is met, such as when a node contains a specified maximum number of objects or reaches a predefined level of depth.

Each node in a quad-tree holds data related to the spatial region it represents, such as the objects located within that region or references to further subdivided regions. The recursive division of the space enables efficient searching, insertion, and deletion of spatial data. A key feature of the quad-tree is its ability to quickly prune large portions of the search space, making it highly effective for spatial range queries, nearest neighbor searches, and point location queries, where the goal is to find all points within a certain area or the closest point to a given location.

Quad-trees are commonly used in applications such as geographic information systems (GIS), computer graphics, image processing, and spatial databases. They are especially effective in situations where the spatial data is sparse in some areas but dense in others, as they allow the space to be divided more finely where necessary while keeping the overall structure relatively simple.

A k-d tree [35] is a data structure used for organizing points in a k-dimensional space, where k can be any number. It is a binary tree where each node represents a point in the k-dimensional space, and the tree recursively partitions the space into two halves at each level based on one of the dimensions. This makes k-d trees particularly useful for multidimensional searching, such as range queries, nearest neighbor searches, and other spatial queries that involve high-dimensional data.

In a k-d tree, each level of the tree corresponds to one of the dimensions in the k-dimensional space. For example, in a 2D space (k=2), the first level of the tree might split the points based on the x-coordinate, and the second level would split based on the y-coordinate. At each level, the points are divided into two subsets: one subset contains points that are less than the median value of the current dimension, and the other subset contains points greater than the median value. This process continues recursively, alternating dimensions at each level until the tree reaches a desired depth or a stopping condition, such as having a small number of points at the leaf nodes.

The structure of the k-d tree allows for efficient searches, especially for queries like nearest neighbor search, where the goal is to find the closest point(s) to a given query point. The recursive partitioning allows for a search that quickly eliminates large portions of the space that are irrelevant to the query, improving performance significantly compared to a brute-force search.

Grid based indexing is commonly used to partition spatial data by dividing the space into a regular, uniform grid of cells. Each spatial object is then assigned to all cells that the object overlaps. By partitioning the data into small, manageable regions, grid-based indexing helps improve the efficiency of spatial operations, such as range queries, making them a popular choice due to their efficiency in searches and updates [36, 37, 38]. Since, the space is divided into spatially disjoint regions, objects that overlap with multiple partitions need to be replicated in each of them. As a result, their performance can be hindered by requiring a secondary filter step to avoid duplicate results, and they require more storage space. To address this issue, a secondary partitioning approach has been proposed [39], which eliminates the generation of duplicate results and, consequently, removes the need for a filtering step. This method classifies objects within each partition into four categories based on whether they start before or after the partition in each dimension. When a range query is evaluated, only a subset of object classes in each partition is selected, ensuring that the query results are free of duplicates.

2.2 Indexing metric spaces

Indexing high-dimensional spaces is a hard problem for two main reasons. First, due to the curse of dimensionality [40], if data points are uniformly distributed, the probability that two points are too close or too far from each other is very low, rendering similarity search mostly meaningless. However, in many real applications, data typically form clusters, so this predicament does not apply. Second, indices that divide the data space into orthotopes (e.g., the R-tree [31], the KD-tree [35], etc.) do not perform well, as they necessarily use only a limited number of dimensions,[1] hence orthotopes end up spanning the entire domain on most dimensions and do not sepa-

[1]Partitioning a D-dimensional space at least once in each dimension defines 2^D partitions, which are much more than the data points for large D (e.g., $D = 100$).

rate the objects well. Besides, such indices are only applicable in vector spaces and are mostly suited for hyperrectangular range queries rather than distance-based search. In low-dimensional spaces, rectangular range queries are often sufficient to find the relevant data points, as the data are more easily separated by simple boundary conditions. However, in high-dimensional spaces, this type of query becomes less effective. Instead of looking for points within fixed boundaries, we typically perform similarity searches based on distance metrics, where the notion of proximity becomes much more complex. As the number of dimensions increases, the notion of a ”rectangular” region becomes less meaningful, and the structure of the data makes traditional range queries increasingly impractical for capturing the true relationships between data points.

Ref. [41] (see also [42]) is a recent comprehensive survey of existing indices for exact similarity search in metric spaces. Based on this study, *pivot-based* indices are the most effective ones. These methods select few *vantage* points (a.k.a. *pivots*, *landmarks*, *representatives*), partition the data space based on them, and use the vantage points to prune the search space, guided by the triangle inequality. Pivot-based indices are applicable even when distances are not computed using point coordinates, but in arbitrary metric spaces (e.g., as shortest paths in graphs [43]). Besides being very efficient, pivot-based indices provide exact and explainable results to similarity queries, which is imperative in application domains like public safety [41], bioinformatics [44], and computer forensics [45]. Hence, performing exact search in the original metric space may be preferable over solutions that transform the data to a vector space using machine-learning techniques (e.g., embedding approaches [46]) and apply search in the transformed space or LSH-based approximate indices [47, 48, 49, 50]; the latter are mostly appropriate in spaces where objects are not well-separated, thus exact similarity search may not be meaningful or critical.

While approximate similarity search methods are commonly used when the application allows sacrificing accuracy for speed, they are typically employed in scenarios where large datasets make exact search computationally expensive or when it is not crucial for the application to provide exact results. These methods prioritize speed and efficiency by providing near-optimal results rather than guaranteed exact answers. They are particularly useful when some degree of imprecision is acceptable, especially in situations where objects are not well-separated in the space, and a small margin of error in the results won’t critically impact the application.

However, approximate methods may compromise the quality of results, leading to reduced precision and potentially missing highly relevant data points. For example, in the field of medicine, where precise matching of genetic data or tumor markers is essential for accurate diagnoses and treatment plans, approximate methods could lead to significant errors with critical consequences.

For our purposes, we focus on exact similarity search because we target applications with smaller datasets, where computational resources allow for the use of precise methods without the performance bottlenecks often associated with larger datasets. In these contexts, the accuracy of results is crucial, and the computational cost of exact search is manageable.

We present in detail three representative main-memory pivot-based metric indices that we use as competitors to our proposed adaptive index for metric spaces, described in Chapter 3.

SimplePivot employs Farthest First Traversal (FFT) [51] to choose vantage points. The FFT algorithm starts by selecting a random point as the first pivot. In each subsequent iteration, it selects as a pivot the object u that maximizes $\min_{p \in P} d(u, p)$, where P is the set of previously selected pivots. Thereafter, we compute the distances of each data point to *all* vantage points. When evaluating queries, we use the precomputed distances to avoid unnecessary distance computations. Specifically, consider a query point q and radius ϵ. At the beginning of query evaluation, we compute and cache $d(p_i, q)$ for all p_i. For each data object $o \in S$, if there exists a pivot p_i, such that $|d(p_i, o) - d(p_i, q)| > \epsilon$, then o is certainly not a result, hence we do not need calculate $dist(q, o)$. Since $d(p_i, o)$ has been precomputed, the pruning test for each object o takes $O(m)$ time, where m is the number of pivots.

The *Spatial Approximation Tree* (SAT) [52] is a hierarchical data structure, where the children of a node are its neighbors in the Delaunay graph on the entire data set. To find the nearest neighbor (NN) of a query object q, we start from the root n; if n is closer to q than its children, then search stops reporting n as the NN. Otherwise, we navigate to the child of n nearest to q and search recursively. To evaluate a range query (q, ϵ), SAT uses the triangle inequality to prune nodes (and corresponding sub-trees) that are guaranteed to be further than ϵ from q.

The *Vantage Point tree* (VP-tree) [53] partitions the data hierarchically based on a *vantage point* at each node. Starting with the root, which indexes all data, at each node v, a vantage object (pivot) p_v is selected at random from the objects indexed at this

node. The data under node v are then split in two partitions as follows: Let μ be the mean distance of points under v to pivot p_v. Objects having distance to p_v less than or equal to μ are placed in the *left* sub-tree; remaining objects go to the *right* sub-tree. To evaluate a range query, we recursively traverse the VP-tree. For a query (q, ϵ), at each node v with pivot p_v having median distance from its indexed points μ, we examine the following:

- if $dist(q, p_v) \leq \epsilon$, then p_v is a result;
- if $|dist(q, p_v) - \mu| \leq \epsilon$, then search the *left* sub-tree;
- if $dist(q, p_v) + \epsilon > \mu$, then search the *right* sub-tree.

For a single query, we may follow multiple paths of the VP-tree, according to the above. The MVP-tree [54] generalizes the VP-tree to a m-ary tree. Instead of splitting the objects in two partitions, it orders them by their distance to the vantage point and partitions them to m groups of equal cardinality. During search, it uses the mean distance for each of the m groups to prune sub-trees that cannot include query results. According to the extensive experimental study in [41], the MVP-tree performs best compared to a wide-range of main-memory metric space indices, including pivot-based methods [55, 56] and SAT [52]. Notable metric-space indices optimized for secondary memory are the M-tree [57] and the PM-tree [58]

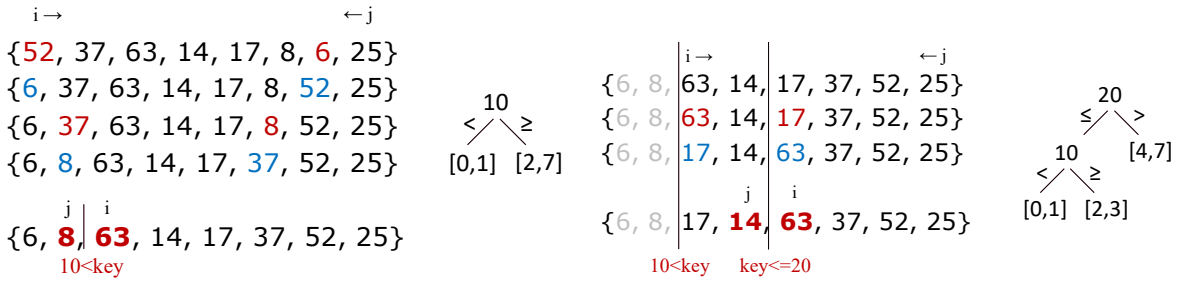
2.3 Adaptive Indices

Adaptive indexing refers to the approach where an index is not constructed apriori for a set of data. Instead, the index is built dynamically and progressively as queries are executed and results are retrieved. This technique first appeared under the name *Deferred Data Structuring* [59]. The concept has since evolved and been reintroduced in various forms, such as database cracking [1, 5, 17, 25, 26, 60] and progressive merging [61], leading to hybrid versions [23, 62, 63].

2.3.1 Database Cracking

Adaptive indexing constructs a data structure for a static dataset on demand by adapting to an evaluated query workload [59]. Each query divides the data space based

on its results, triggering the construction of a search tree that guides the evaluation of subsequent queries. This idea has been applied to database indexing [1], progressively *cracking* an initially unsorted array to segments that obey a total order and constructing a binary search tree to prune sub-arrays that do not contain query results. Figure 2.1 shows the cracking of an unsorted array based on query $10 < x \leq 20$ and one step in the progressive construction of the corresponding binary search tree. The array is first cracked based on $10 < x$. Indices i and j scan the array in forward and backward, respectively. For each out-of-order value found (i.e., a value > 10 at i and one ≤ 10 at j), if $i < j$, the values are swapped and the process continues; otherwise, cracking stops. After cracking based on $10 < x$, we update the binary search tree: all values less than 10 are in array positions 0 to 1 and all values of 10 or larger are in array positions 2 to 7. The second crack, based on $x \leq 20$ applies on the right child of the root and cracks the corresponding subarray to two pieces: positions 2 to 3 having keys less than or equal to 20 and positions 4 to 7, having keys greater than 20. The search tree (e.g., an AVL-tree) is re-balanced after each cracking.



(a) cracking based on $10 < x$

(b) cracking based on $x \leq 20$

Figure 2.1: Standard cracking example

In effect, cracking conducts *quicksort* incrementally, triggered by queries. This process has been extended to efficiently handle updates efficiently [30]. As an alternative to quicksort, one may perform a *mergesort* operation incrementally [61], a hybrid approach that combines merging and cracking [63], or first partition the domain in disjoint ranges and then crack the partitions [26].

Cracking variants

Database cracking is extended to address its limitations, namely CPU efficiency, convergence, tuple reconstruction, and robustness.

Hybrid cracking [63] is introduced to improve the convergence of standard cracking to a full index. It combines ideas from adaptive merging [61] with database cracking to achieve faster convergence while maintaining low initialization costs. The main drawback of standard cracking is that it can only create two new partitions per query, requiring multiple queries to reach a full index. In contrast, adaptive merging creates initial sorted runs, which incurs a high cost for the first query.

Hybrid cracking addresses the challenges of slow convergence by creating initial unsorted partitions that fit in memory, which are then physically reorganized and adaptively merged. As queries are processed, qualifying tuples are moved from the initial partitions into final partitions. Each initial partition uses a table of contents to track the key ranges it contains, and a master table of contents is used to track both initial and final partitions. As tuples are transferred, both tables are updated. Lightweight reorganization is applied to the initial partitions, and several strategies for reorganizing both initial and final partitions—such as sorting, cracking, and radix clustering—are explored.

For the initial partitions, cracking proves more effective than radix clustering, as radix clustering introduces some overhead in the first query. Sorting, on the other hand, does not work well for the initial partitions due to the high overhead it adds to the first query. For the final partitions, both radix clustering and sorting are viable, depending on the scenario. Radix clustering offers a lightweight footprint for scan queries and achieves optimal performance quickly, while sorting is slower for the first query but achieves optimal performance faster. By using cracking for the initial partitions and sorting for the final partitions, hybrid cracking achieves better convergence. The implementation of hybrid crack sort demonstrated superior performance compared to standard cracking, full index, and scan, showing faster convergence.

Sideways cracking [64] addresses the inefficiency of tuple reconstruction in standard crackinging cracker maps. A cracker map consists of two logical columns: the cracked column and a projected column. It ensures that the projection attributes remain aligned with the selection attributes. When a query is processed, sideways cracking creates and cracks only those cracker maps that contain any of the accessed attributes, ensuring that each accessed column is aligned with the cracked column of its corresponding cracker map. If the attribute access pattern changes, the cracker maps may reflect different progressions based on the applied cracks. Sideways cracking maintains a log to track the state of each cracker map and synchronize them

when necessary. This approach allows sideways cracking to function without needing prior knowledge of the workload and adapts the cracker maps to the attribute access patterns. Additionally, it improves efficiency and reduces overhead by only materializing the parts of the projected columns in the cracker maps that are actually queried, known as partial sideways cracking.

Stochastic cracking [17] addresses performance unpredictability in database cracking by introducing additional random cracks during query time to help partition a column more uniformly. A key issue with standard cracking is that partition boundaries are highly dependent on the query sequence. This can result in unbalanced partitions, and subsequent queries may need to reorganize large chunks of data. To mitigate this, stochastic cracking adds random cracks alongside the query-driven cracks, evolving the cracker index more evenly.

Several variants of stochastic cracking have been proposed to introduce these additional cracks, including data-driven and probabilistic approaches. These variants strike a balance between the variance they introduce and the overhead associated with the cracking process. One such approach is the Data Driven Center (DDC) algorithm, which recursively halves the portion of the array where the query range falls until the piece is sufficiently small. However, finding medians for halving is computationally expensive, leading to significant overhead. Another approach, the Data Driven Random algorithm (DDR), selects random pivots until the target piece becomes smaller than a set threshold. Both methods, though effective, incur considerable overhead from the auxiliary operations.

To reduce this overhead, variants such as DD1C and DD1R were introduced, performing at most one auxiliary reorganization. DD1C halves a piece and cracks the remaining part, while DD1R only performs a single random reorganization. Despite this reduction in auxiliary operations, the cost of even a single reorganization can still add significant overhead, especially during the first few queries. The MDD1R variant, which further reduces initialization cost, addresses this by avoiding cracking based on query bounds. Instead, the results are materialized in a new array, much like a standard select operation, and integrated with a random crack on another piece to prevent an additional scan.

This MDD1R variant has shown the best overall performance, as it cracks partitions at the query boundaries with one random split, materializes the result of each boundary partition in a separate view, and builds the result by reading from the

index for inner partitions and from the views for boundary partitions.

2.3.2 Multidimensional Adaptive and Progressive indices

The *QUery-Aware Spatial Incremental Index (QUASII)* [19] was the first attempt to apply adaptive indexing to spatial data. It organizes objects based on each query's lower and upper coordinates by slicing each dimension and performing nested reorganizations. The process begins by reorganizing objects along the x-dimension, producing three slices. Next, the middle slice from the x-dimension is reorganized along the y-dimension, creating three slices, with the middle one containing objects in the range [xl, yl]. Finally, the middle slice from the y-dimension is reorganized along the z-dimension, resulting in a slice that contains the query result. This hierarchical structure becomes more granular as we move deeper into the index. Figure 2.2 illustrates the indexing strategy employed by QUASII.

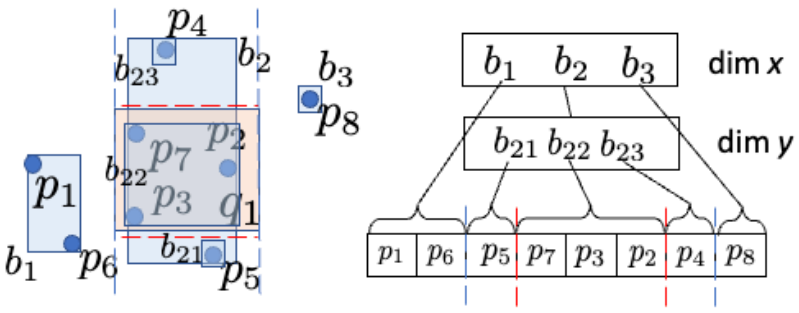


Figure 2.2: QUASII indexing strategy.

Unlike a KD-tree, which splits the space on the same dimension at each level, QUASII cracks one dimension per tree level. It first cracks along the x-dimension to organize the data at the first tree level, then cracks the piece corresponding to the query's x-range along the y-dimension at the second level, and so on. Each level in the resulting wide tree corresponds to one of the d dimensions, with each level focusing on progressively refining the query's range. During query processing, the hierarchical structure is traversed depth-first, and additional refinements are performed when the size of a slice exceeds a defined threshold. This incremental approach to indexing allows QUASII to adaptively organize spatial data in response to specific queries.

The *Adaptive KD-tree* (AKD) [18, 20] is a dynamic extension of the traditional KD-tree that adapts its structure in response to the queries it processes, making it particularly well-suited for environments with fluctuating or unpredictable query

patterns. Unlike the conventional KD-tree, where data is statically partitioned along alternating dimensions at fixed points, the AKD refines its partitioning dynamically during query execution. As queries are issued, the tree modifies its structure to align with the specific access patterns of those queries. For instance, if a query targets a particular region of the data space, the tree can further partition that region, allowing it to optimize indexing in real-time based on the evolving query workload. This adaptability enables the AKD to focus on regions of the dataset that are frequently accessed, improving query performance over time.

The AKD processes all lower bounds of a query before handling the higher bounds and assigns dimensions to tree levels in a round-robin manner, ensuring balanced partitioning. Unlike QUASII, which cracks the data along a single dimension at each tree level, the AKD cracks the same dimension multiple times at different levels, preserving its binary tree structure throughout. This strategy allows for more granular and responsive partitioning as the data and queries evolve. While the AKD offers a high degree of flexibility and responsiveness to query patterns, it can also incur high initial query costs. Specifically, it may require substantial partitioning effort before it can efficiently handle queries, particularly during the early stages of index construction. This overhead, especially for the first few queries, can affect its overall performance in scenarios where initial query costs are critical. Figure 2.3 shows the partitioning strategy of AKD.

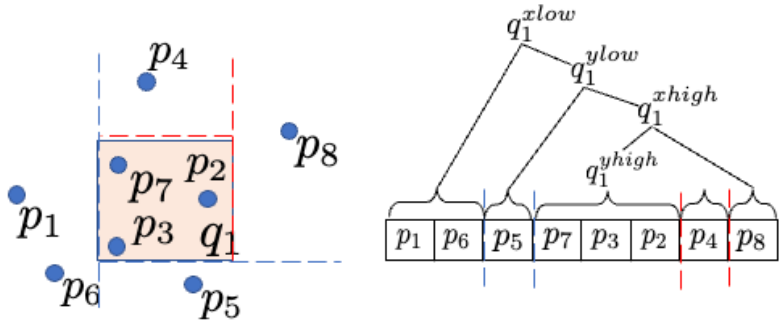


Figure 2.3: AKD indexing strategy.

The *Progressive KD-tree* [18] takes a more controlled approach to partitioning compared to the Adaptive KD-tree. This structure allows indexing to happen incrementally, which is particularly useful when there is a need to balance the time and resources spent on index refinement versus query processing. Rather than continually

refining the index, the Progressive KD-tree uses a defined budget for each query to decide how much to invest in further index refinement. As queries are processed, the index is progressively improved, but only as much as needed based on the current workload. This prevents unnecessary refinement and ensures that query performance is maintained without burdening the system with excessive index construction. The Progressive KD-tree adapts to the query workload by adjusting the budget for refinement dynamically, providing an efficient balance between indexing effort and query response time.

The *Greedy Progressive KD-tree* [18] is a specialized variant of the Progressive KD-tree, where the index refinement process is governed by a greedy algorithm. This algorithm prioritizes which parts of the data to refine based on the potential improvements in query performance. Instead of refining all regions of the dataset uniformly, the Greedy Progressive KD-tree focuses on the most important areas—those that are expected to yield the greatest benefit in terms of query response time. By selecting these critical regions for refinement, the Greedy Progressive KD-tree optimizes its resources and achieves faster query performance. This greedy approach ensures that the system does not waste resources on less critical regions of the data, making it more efficient than the standard Progressive KD-tree. However, the greedy strategy assumes that query patterns are somewhat predictable or stable, as it relies on prior query behavior to make decisions about which parts of the tree to refine.

The Greedy Progressive KD-tree (GPKD) takes the approach of the PKD a step further by using a cost model to estimate the execution time of each query. This model helps ensure that the execution time remains consistent and robust during the index growth phase. By prioritizing index refinement in areas that offer the most significant performance gains, the GPKD seeks to optimize query performance while maintaining a steady index construction process. The GPKD's cost model allows it to dynamically adjust the indexing effort based on the expected benefits, providing a more efficient use of resources compared to the standard PKD.

Those approaches were recently superseded by the *adaptive incremental* R*-tree* (AIR) [21], which builds a compact tree by overseeing all dimensions in each index level and applying quality-aware criteria in splitting and adjusting tree nodes based on query boundaries. AIR commences with an unorganized static data array and progressively organizes queried data areas while responding to queries. It initially comprises a single leaf root enclosing all the data, relaxing the principles of a

traditional R-tree, whereby each node holds no more than a predefined number of entries. Leaves of cardinality above a threshold M_ℓ, called *irregular*, are eligible for cracking, while those below M_ℓ, called *regular*, are not cracked further. As the example in Figure 2.4 shows, in response to a range query, represented by a rectangle, AIR *cracks* the data space through $2d$ hyper-planar cracks (dashed lines), each yielding a spatial partition, with a remaining partition containing the query results; each crack reorganizes the data array accordingly, as the bottom part of Figure 2.4 shows. To avoid the repercussions of a pathologically skewed workload, AIR adds a *stochastic* crack on the largest ensuing piece, totaling at most $2d + 2$ pieces in d dimensions. We illustrate this process for a single node fully containing a query range, yet it is applicable on each leaf node that overlaps the query. AIR cracks *irregular* leaves in response to queries, eventually creating *regular* leaves, and evolves into a structure resembling a classic R-tree that outperforms prior multidimensional adaptive indexing methods [28, 19] across spatial and multidimensional point datasets. Figure 2.4 illustrates the cracks formed by AIR's indexing strategy, as well as the reorganization of the data array.

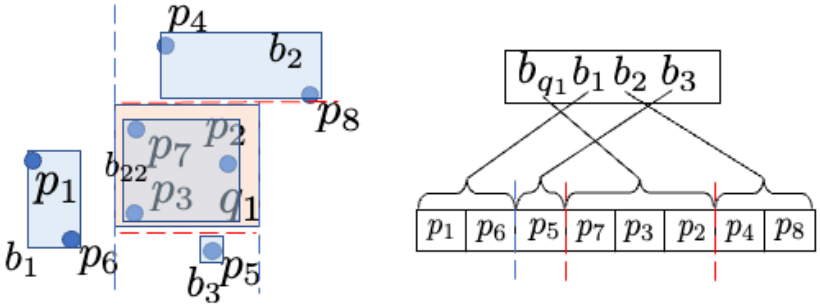


Figure 2.4: AIR indexing strategy.

2.4 Learned Indices

Learned indices leverage machine learning (ML) models to replace traditional index structures like B-Trees, hash indices, and bitmaps. These models map keys to positions or conditions, offering more optimized index structures that reflect data patterns more efficiently than traditional approaches. Traditional indexing methods, such as B-Trees, are designed for general-purpose data access patterns but may not fully exploit the distribution of the data. Learned indices, however, adapt to data

distributions by automatically synthesizing specialized indexing structures, improving memory efficiency and query performance, especially in cases with predictable or evolving data patterns.

In [65], the authors propose the Recursive Model Index (RMI), which uses a hierarchy of models to predict key locations, replacing B-Tree internal nodes with models. The Naïve learned index uses the cumulative distribution function (CDF), but RMI refines this approach with multiple prediction levels and error bounds. The piecewise linear function-based approach in [66] approximates data distribution with a bounded error and stores segments in a B+-Tree, optimizing memory usage and performance.

ALEX [67] builds on RMI, supporting updates via exponential search and using linear regression models at inner nodes. Its gapped array layout optimizes space and performance, especially during insertions. The PGM-index [68] combines probabilistic graphical models with worst-case performance guarantees for efficient insertions and deletions, making it suitable for real-time applications. In [69], a learned indexing structure for multi-dimensional data adapts the index for better performance in space and query efficiency.

RadixSpline [70] combines Radix trees with spline interpolation to optimize range queries on high-dimensional data, while Tsunami [71] leverages deep learning to predict key locations in multi-dimensional data with skewed and correlated distributions. It outperforms traditional multi-dimensional indices by reducing memory usage and improving query times. Hist-Tree [72] challenges learned indices by leveraging implicit assumptions, such as sortedness and range, using histograms at each node to better represent key distribution, outperforming learned indices in certain scenarios.

Several learned index structures have been proposed [73, 74, 75, 76, 77, 78, 79], with benchmarks offering insights into their performance [80, 81, 82] and surveys providing a clear picture into their strengths and weaknesses [83, 41]. Additionally, learned cardinality estimation has emerged, employing ML techniques to estimate the number of distinct elements in datasets or data streams [84, 85, 86].

Learned indexes have a significant start up cost, which is even higher than that of classic indexes. Therefore, in our application scenarios where the objective is to minimize the cumulative cost of index building plus query processing, learned indexes may not stand as viable competitors to adaptive ones.

Chapter 3

Adaptive Indexing in High-Dimensional Metric Spaces

Similarity search in high-dimensional metric spaces is a common technique used in various applications, such as content-based image retrieval, bio-informatics, data mining, and recommender systems. To speed up search processes, indices are often employed. However, building an index for high-dimensional spaces can be computationally expensive, and it may not be cost-effective if the number of queries is low. In these cases, constructing an index adaptively, in response to an evolving query workload, becomes more advantageous.

Indexing high-dimensional spaces presents a significant challenge for two primary reasons. First, due to the curse of dimensionality, if data points are uniformly distributed, the likelihood that two points are either too close or too far apart is very low, making similarity search ineffective. Second, traditional indexing methods, such as the R-tree and KD-tree, divide the space into orthotopes (hyperrectangular units). These methods struggle in high-dimensional spaces because they can only utilize a limited

number of dimensions, causing the orthotopes to span the entire domain across most dimensions and failing to effectively separate the objects. Furthermore, such indices are often limited to vector spaces and are generally optimized for hyperrectangular range queries, rather than distance-based searches.

In this chapter, we propose a method for adaptive high-dimensional indexing, the AV-tree, that exploits previously computed distances, using query centers as vantage points. The AV-tree differs from VP-tree variants in the following ways: (i) it builds the tree *progressively*, via query evaluation, rather than in advance; (ii) it selects pivots *adaptively* from queries rather than from the data collection; and (iii) it keeps the data in a *single array* and swaps them to partition them to sub-arrays.

Outline. In Section 3.1, we introduce the definitions and foundational concepts essential for understanding the methodology presented in this work. Section 3.2 provides a comprehensive overview of our indexing and querying algorithms, including enhancements to improve performance. Section 3.3 discusses the experimental setup and presents the results of our experiments, evaluating the effectiveness of our approach. Finally, Section 3.4 offers a summary of the key findings and concludes with potential directions for future research.

3.1 Definitions and Preliminaries

We examine similarity-based queries in a metric space. A metric space is defined as a pair $\{M, d\}$. M is a domain wherefrom objects are instantiated. For example, if M is be a D-dimensional vector space, each object o in it has the form $\langle id, o_1, o_2, \ldots, o_D \rangle$, where id is an identifier and $o_i \in [0, 1]$ is the i-th dimensional value of o; d is a metric distance function applied between objects in M; in vector spaces, d is typically the Euclidean distance, i.e., $d(q, o) = \sqrt{\sum_{i=1}^{D} (q_i - o_i)^2}$. In general, a metric distance function d has the following four properties:

- **Identity.** The distance of an object to itself is 0; $d(x, x) = 0$.
- **Non-negativity.** The distance between two distinct objects is positive; if $x \neq y$ then $d(x, y) > 0$.
- **Symmetry.** The distance from x to y is the same as that from y to x; $d(x, y) =$

$d(y, x)$.

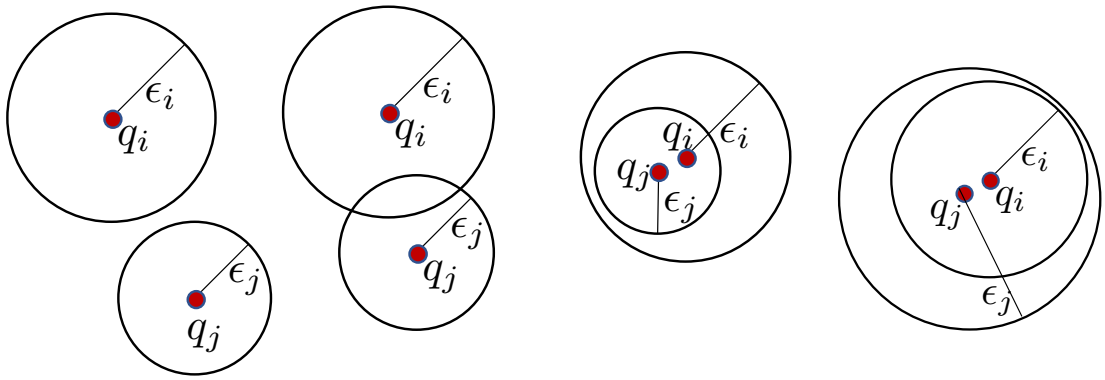
- **Triangle Inequality.** For any three objects x, y, z, $d(x, z) \leq d(x, y) + d(y, z)$.

We consider a set O of objects in the metric space M. The most common similarity-based queries are the range query and the k-nearest-neighbor query.

Definition 3.1. Range Query. Given an object q and a distance bound ϵ, a range query returns all objects $o \in O$ that are within distance ϵ from q, i.e., $d(q, o) \leq \epsilon$.

Definition 3.2. k-Nearest-Neighbor (kNN) Query. Given an object q and a positive integer k, $k \leq |O|$, a k-nearest-neighbor query returns a subset $R \subseteq O$, such that $|R| = k$ and $\forall o \in R, o' \in O \setminus R : d(q, o) \leq d(q, o')$; in other words, a kNN query finds a set R of k objects in O having no larger distances to q than those outside R.

Answering a query amounts to finding the identifiers of the objects $o \in O$ in the result set. We assume that the objects are stored row-wise in memory, i.e., the entire tuple of the first object precedes the tuple of the second object, and so on; such a representation facilitates efficient distance computations.



(a) disjoint (b) overlap (c) q_i contains q_j (d) q_j contains q_i

Figure 3.1: Four cases of overlap between q_i and q_j

We aim to exploit previously evaluated queries to expedite the evaluation of subsequent queries. To do so, using the triangle inequality, we quickly determine whether, and how, the range of a previous query (q_i, ϵ_i) overlaps with that of the current query (q_j, ϵ_j). There are four cases in this regard, depicted in Figure 3.1.

(a) (q_i, ϵ_i) does not overlap (q_j, ϵ_j), i.e., $d(q_i, q_j) > \epsilon_i + \epsilon_j$.

(b) (q_i, ϵ_i) overlaps (q_j, ϵ_j), but neither is a subset of the other.

(c) (q_i, ϵ_i) contains (q_j, ϵ_j), i.e., $\epsilon_i \geq d(q_i, q_j) + \epsilon_j$.

(d) (q_j, ϵ_j) contains (q_i, ϵ_i), i.e., $\epsilon_j \geq d(q_i, q_j) + \epsilon_i$.

3.2 The AV-tree

In this section, we present our proposed *adaptive vantage* tree (AV-tree) for high-dimensional metric spaces. We first show how the AV-tree is incrementally constructed while evaluating range queries and then present the corresponding algorithm for kNN queries. Notably, range queries can be interleaved with kNN queries in a mixed workload without affecting the data structure and its effectiveness. In Section 3.2.3, we present some enhancements to the basic version of our index, followed by a cost analysis (Sec. 3.2.4).

3.2.1 Range Query

Our algorithm builds on the framework for one-dimensional database cracking with some significant departures: First, there is no total order of the indexed data to guide the process. Second, contrary to existing multidimensional cracking approaches [19, 18, 21], we partition the space based on the distances between the query and data, rather than using hyperrectangular partitions.

We evaluate the first query (q_1, ϵ_1) by scanning the entire data array O, and, while computing the results, performing a *crack-in-two* operation: we place data points $o \in O$ with $d(q_1, o) \leq \epsilon$ before data points $o \in O$ with $d(q_1, o) > \epsilon$. At the same time, we define the root of the *adaptive vantage* tree, or AV-tree, a binary search tree which helps identify relevant data for subsequent queries and avoid redundant computations. For each subsequent query, we use the AV-tree to guide search and expand it by introducing new cracks.

Each node v in the AV-tree contains two elements: the *scope* $[v.lo, v.hi]$ of v, i.e., the range of array indices that v indexes; and the *query* $(v.q, v.\epsilon)$, that guides the search in v, if v is not a leaf node. The tree root has $lo = 0$ and $hi = |O| - 1$, where $|O|$ is number of data objects. If v is a leaf, then $v.q$ is null. Otherwise, v has two pointers $v.\mathsf{left}$, $v.\mathsf{right}$ to its left and right children, respectively. For each object o in the scope of $v.\mathsf{left}$, it is $d(q, o) \leq v.\epsilon$, while for each object o in the scope of $v.\mathsf{right}$,

it is $d(q, o) > v.\epsilon$.

Algorithm 3.1 presents the search-and-crack process in detail, outlined in two procedures. The main recursive SEARCH-AND-CRACK procedure takes as input an array O with the data points, a query point q and the corresponding distance bound ϵ and the node v of the AV-tree on which it is applied. For a new query, we initialize the query result to $R = \emptyset$ and call the procedure with v being the tree root. If $v.q$ is null, then v is a leaf node, hence we crack by procedure CRACK-IN-TWO (described later), yielding two new vertices as children of v. If node v is not a leaf, then we examine the relationship between the node query range $(v.q, v.\epsilon)$ and the new query range (q, ϵ) as in Section 3.1. If the two ranges are disjoint, all data under the scope of $v.\mathsf{left}$ are not part of the query result, hence we call SEARCH-AND-CRACK for the right child $v.right$, as its scope may include results of q. On the other hand, if the new query range overlaps with $(v.q, v.\epsilon)$, we distinguish two cases. If $(v.q, v.\epsilon)$ is entirely inside (q, ϵ), then we add[1] to R all data under the scope of $v.\mathsf{left}$ as query results and SEARCH-AND-CRACK the right subtree of v. Otherwise, if (q, ϵ) is entirely inside $(v.q, v.\epsilon)$, we only SEARCH-AND-CRACK the left subtree of v. Lastly, if there is no containment relationship between $(v.q, v.\epsilon)$ and (q, ϵ), as in Figure 3.1b, then we also call SEARCH-AND-CRACK for the right subtree.

Procedure CRACK-IN-TWO is based on Hoare's quicksort partitioning [6]; it scans the scope of a node v array O from position lo to position hi and swaps data items to divide $[lo, hi]$ in two parts: $[lo, j]$, including data points o such that $d(q, o) \leq \epsilon$ and $[j+1, hi]$ including remaining points. We add the former part, $[lo, j]$, to the query result R and generate two new nodes for the two new scopes, as children of calling node v. Note that one of the two scopes may be *empty*, in case the scope of v includes either (i) no query results or (ii) only query results. In the former case, $v_L.lo = lo$ and $v_L.hi = lo - 1$; in the second case, $v_R.lo = hi + 1$ and $v_L.hi = hi$. Procedure SEARCH-AND-CRACK does not perform recursive calls for a child having empty scope, as no query results can be obtained from such nodes. We call leaves that cannot be further cracked because they have an empty scope *empty* leaves.

Example. Figure 3.2 presents a detailed example. Data array O includes eight 2D points, p_1 to p_8, and initially the tree has a single node v_1 with scope $[0, 7]$. Upon the first query, (q_1, ϵ_1), CRACK-IN-TWO runs for the root node v_1 (Line 3), which is a

[1]For simplicity, we denote the query result as a set of interval ranges indicating the positions of result objects in array O.

Algorithm 3.1 Distance-Range Search and Crack

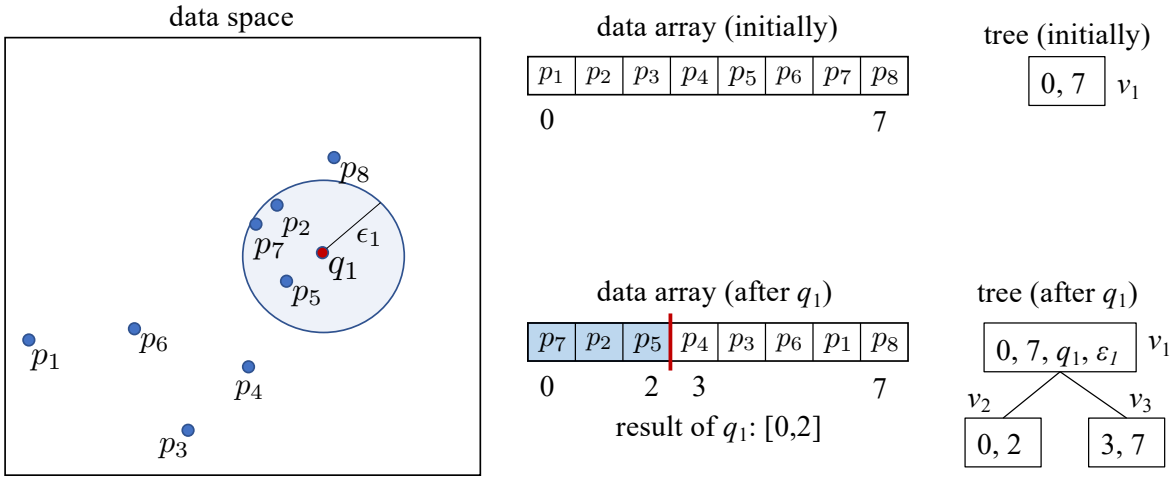
```
1: procedure SEARCH-AND-CRACK(data array O, query q, bound ε, node v, result R)
2:     if v.q is null then                                              ▷ leaf node
3:         (v.left, v.right) ← CRACK-IN-TWO(O, v.lo, v.hi, q, ε, R)
4:         v.q ← q; v.ε ← ε
5:     else                                                             ▷ non-leaf node
6:         if d(q, v.q) > ε + v.ε then                                  ▷ disjoint query ranges
7:             SEARCH-AND-CRACK(O, q, ε, v.right, R)
8:         else                                                         ▷ overlapping query ranges
9:             if d(q, v.q) < ε − v.ε then                              ▷ v.q entirely inside q
10:                R ← R ∪ [v.left.lo, v.left.hi]                       ▷ update query result
11:                SEARCH-AND-CRACK(O, q, ε, v.right, R)
12:            else
13:                SEARCH-AND-CRACK(O, q, ε, v.left, R)
14:                if d(q, v.q) ≥ v.ε − ε then                          ▷ q not entirely inside v.q
15:                    SEARCH-AND-CRACK(O, q, ε, v.right, R)
16:
17: procedure CRACK-IN-TWO(array O, int lo, int hi, query pt q, bound ε, result R)
18:     i ← lo
19:     j ← hi
20:     while true do
21:         while d(q, O[i]) ≤ epsilon and i ≤ hi do
22:             i ← i + 1
23:         while d(q, O[j]) > epsilon and j ≥ lo do
24:             j ← j − 1
25:         if i ≥ j then
26:             break
27:         swap O[i] with O[j]                                          ▷ O[i] and O[j] on wrong sides
28:     R ← R ∪ [lo, j]                                                  ▷ update query result
29:     v_L.lo ← lo;   v_L.hi ← j;   v_L.q ← null
30:     v_R.lo ← j + 1;   v_R.hi ← hi;   v_R.q ← null
31:     return (v_L, v_R)
```

leaf, to produce two new nodes, v_2 and v_3, as its left and right child, respectively, as Figure 3.2a shows. The result of R is the scope of v_2 (i.e., p_7, p_2, and p_5). For the second query, (q_2, ϵ_2), SEARCH-AND-CRACK runs for $(q, \epsilon) = (q_2, \epsilon_2)$. Figure 3.2b shows that the range of q_2 overlaps with the query range of the root, (q_1, ϵ_1). Hence, we enter Lines 13–15. The call in Line 13 yields a CRACK-IN-TWO of v_1's left child (i.e., node v_2). However, this crack produces no results, because all points in the scope of v_2

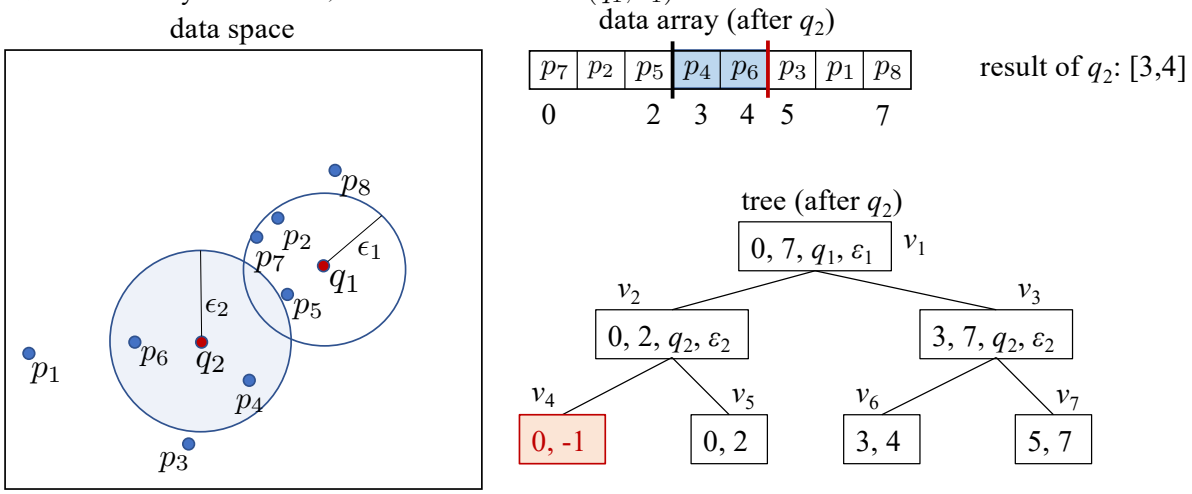
are outside q_2's range; thus the newly produced node v_4 as left child of v_2 has *empty range* [0,-1] (shaded in the figure). The generated right child v_5 has the same scope as its parent v_2. The recursive call at Line 15 invokes CRACK-IN-TWO for v_1's right child (i.e., v_3). Now we do have a query result [3,4] added to R (i.e., points p_4 and p_6) and two new vertices v_6 and v_7 with non-empty scopes as new children of v_3. Figure 3.2c shows the effect of the next query, (q_3, ϵ_3). Query range (q_3, ϵ_3) is outside the range of the root v_1, hence we only visit its right child v_3 (Line 7). We find that (q_3, ϵ_3) is fully contained in (q_2, ϵ_2), hence only visit v_3's left child v_6 (Line 13), where we find no results for q_3, yielding v_8 as an empty child of, and v_9 as identical to, v_6.

3.2.2 Nearest-Neighbor Query

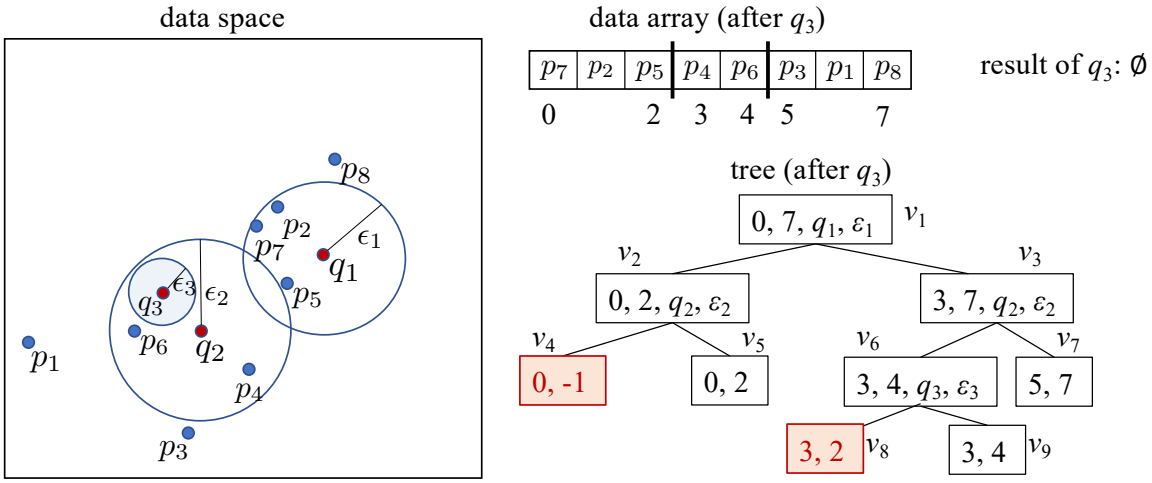
Like range queries, kNN queries crack the data array and progressively construct and use AV-tree as a binary search tree that guides them to relevant data, yet now we access the AV-tree nodes in a *best-first* order that is appropriate for kNN search. Further, adaptive indexing in response to kNN queries poses a distinct challenge, as queries do not readily offer a distance bound. We define such a bound as the distance between the query object and its running k-th nearest neighbor. Algorithm 3.2 presents the process in detail. A query (q, k) comprises a query point q and the integer number of sought nearest neighbors k. We use two priority queues: a min-heap searchPQ that organizes unvisited nodes by least possible distance to q to guide the search in a best-first manner, initialized with the root; and a max-heap resultPQ that holds the running kNN data object results. In each iteration of the while loop (Lines 5–20), we pop the top element v from searchPQ. If v is a leaf (Line 8), we compute the distance of each data object in v to the query point q and update resultPQ accordingly, keeping track of the k nearest objects to q; resultPQ is a max-heap of data-objects ordered by their distance to the query point, hence the top element is the farthest from q, i.e., the running kth Nearest Neighbor. When looping over the data objects in a leaf (Line 9), if one is closer to q than the running kth-NN (Line 10), then we remove the current top element (Line 11), and add this object to resultPQ (Line 12). We also crack the leaf node using as bound (to be enhanced in Section 3.2.3) the distance to the top item in resultPQ (Line 13). If v is an internal tree node (Line 14) then we push its children to searchPQ with their priority key set as the *least possible distance* of an object under v to q, using the triangle inequality. As the left sub-tree



(a) data array and tree, before and after (q_1, ϵ_1)



(b) after (q_2, ϵ_2)



(c) after (q_3, ϵ_3)

Figure 3.2: Search-and-crack example

contains points that are closer than $v.\epsilon$ to v's vantage point, q's minimum possible distance from an object therein is the maximum of 0 and distance of v's vantage point from the query minus $v.\epsilon$ (Line 15); if the latter is negative, then q is in the sphere of v. The right child, conversely, contains the objects outside the sphere centered at v's vantage point with radius $v.\epsilon$, hence q's minimum possible distance from such an object is the maximum of 0 and $v.\epsilon$ minus the distance of v's vantage point from q (Line 18); if the latter is negative, then q is outside the sphere of v. In both cases, if the minimum distance is not smaller than the running k-th smallest distance at the top of resultPQ, then we do not need to look into that child; otherwise (Lines 16 and 19), we add the node to searchPQ (Lines 17 and 20). The search terminates when searchPQ becomes empty.

Algorithm 3.2 kNN Search and Crack

```
 1: procedure kNNSearch(data array O, query q, int k)
 2:     searchPQ ← PriorityQueue⟨dist, node⟩                    ▷ guide search
 3:     resultPQ ← PriorityQueue⟨dist, pid⟩                     ▷ results PQ
 4:     searchPQ.push([0, root])
 5:     while !searchPQ.empty() and
 6:           searchPQ.top().dist < resultPQ.top().dist do
 7:         v = searchPQ.top().deheap()
 8:         if v is leaf then
 9:             for o in v do                                   ▷ o is a data point
10:                 if d(p, q) < resultPQ.top().dist then
11:                     resultPQ.pop()                          ▷ update kNN set
12:                     resultPQ.push([d(p, q), pid])
13:             Crack v                                         ▷ on distance to current k-th NN
14:         else                                                ▷ non-leaf node
15:             leftMinDist = max{0, d(v, q) − v.ϵ}
16:             if leftMinDist < resultPQ.top().dist then
17:                 searchPQ.push([leftMinDist, v.left])
18:             rightMinDist = max{0, v.ϵ − d(v, q)}
19:             if rightMinDist < resultPQ.top().dist then
20:                 searchPQ.push([rightMinDist, v.right])
```

3.2.3 Enhancements

We introduce three enhancements that significantly improve the performance of the AV-tree: First, we crack leaves by the median distance of objects therein to the query.

Second, we eschew constructing empty leaves and cracking leaves that have a few objects. Third, we *cache* and sort the last computed distances of objects in leaves that cannot be cracked further, to avoid distance computations for objects that are definitely not results.

Cracking based on mediocre distances

A popular practice in both cracking-based and pivot-based indexing methods is to partition the data on an intrinsic *median* value rather than an extrinsic threshold. For instance, the VP-tree [53] partitions data on their *median distance* to a vantage point. Likewise, in 1D cracking, using a median or *mediocre* value in a cracked piece rather than a query threshold brings efficiency benefits [29]. Inspired by such precedents, we use a *sample-based mediocre pivot* for leaf cracking. When we crack a sub-array corresponding to a leaf node v, we compute the distances from a few sample points in v to q and crack on the median thereof as $v.\epsilon$. We confirmed experimentally that, with as few as 3 samples, mediocre-based cracking is superior to the default strategy that cracks on the query range ϵ or the running kth NN. Besides, mediocre-based cracking leads to a balanced tree, since each crack yields two partitions of almost equal size.

Avoiding empty leaves and applying a cracking threshold

As we saw in Section 3.2.1, our default algorithm may add *empty leaves* to the AV-tree. Such empty leaves add space and search overhead. In our implementation, we do not create leaves with empty scopes, i.e., we do not commit a crack of a leaf v that results in an empty v_L or v_R and let v remain a leaf. In addition, the default algorithm cracks a leaf v unconditionally; however, leaf nodes that contain few objects are not worth cracking in practice, as they increase tree height without offering a significant pruning advantage. As in 1D cracking [1], we do not crack leaves holding fewer objects than *cracking threshold* θ, which effectively delimits the height of the AV-tree. We thus expect average leaf size to converge to $\theta/2$ and tree height to $1 + \log_2 \frac{|O|}{\theta}$. We call such leaves that are not cracked further *fixed* leaves. When a query reaches a fixed leaf, we obtain query results therefrom by linear scan.

Distance Caching

As distance computations consume most of the query evaluation cost, reducing them is important. While cracking, we can *cache* each object's distance to the current query. Thereby, we get the distance of each object in a leaf v to the leaf's parent node $p(v)$. Next time we visit v while processing another query q, we may use the triangle inequality on $d(p(v), q)$ and the cached distance $d(p(v), o)$ for each object $o \in v$ to check whether o can be a query result and either prune o or add it to the result set R accordingly, avoiding the associated distance computation.

To minimize its overhead, we apply distance caching only in *fixed* leaves; we cache the distance of each object o in leaf v to the leaf's parent node $p(v)$. We *sort* objects by distance to $p(v)$ and follow this order to conduct only a few comparisons to cached distances with early termination, pruning distance computations for objects directly determined to be, or not be, query results.

Algorithm 3.3 shows how we handle fixed leaves using cached distances, modifying procedure SEARCH-AND-CRACK in Algorithm 3.1. We show the differing part for range queries; for kNN queries, we use the distance to the item at the top of the result heap as a threshold in comparisons in place of ϵ and update the kNN result set when accessing qualifying data objects. We now discuss in detail how we prune distance computations in each of four cases, as depicted in Figure 3.3: two for left-child fixed leaves v that point to data within their parent's query range $(p(v), p(v).\epsilon)$ and two for right-child fixed leaves pointing to data outside the query range.

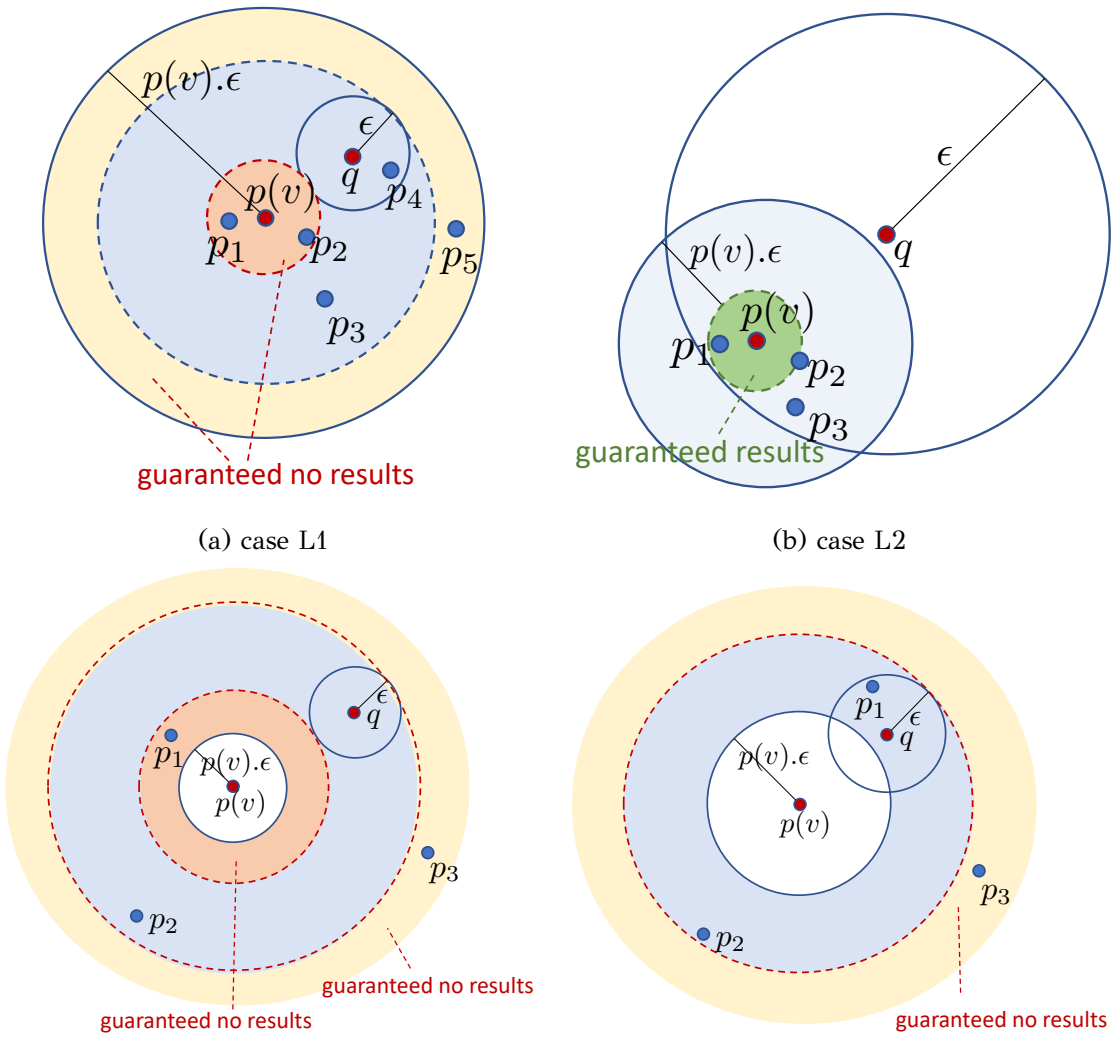
In **Case L1**, $d(q, p(v)) > \epsilon$. Then, by the triangle inequality, objects o with $d(p(v), o) < d(q, p(v)) - \epsilon$ or $d(p(v), o) > d(q, p(v)) + \epsilon$ cannot be query results. We find, by binary search on the sorted cached distances, the position of the first object $o \in v$ with $d(p(v), o) \geq d(q, p(v)) - \epsilon$ and that of the last object $o \in v$ with $d(p(v), o) \leq d(q, p(v)) + \epsilon$, scan objects in-between, and include in the result those having distance to q at most ϵ (Lines 10–14). In Figure 3.3a, assume $p(v)$ is the parent of fixed leaf v, with objects p_1 to p_5 sorted by distance to $p(v)$. Only p_3 and p_4 can be query results; we compute distances only for those two, yielding p_4 as a result.

In **Case L2**, $d(q, p(v)) \leq \epsilon$. Then, objects o with $d(p(v), o) \leq \epsilon - d(q, p(v))$ are surely query results. Hence, we find, by binary search, the position of the first object $o \in v$ with $d(p(v), o) > \epsilon - d(q, p(v))$ (e.g., p_3 in Figure 3.3b), add all objects heretofore to the query result (e.g., p_1 and p_2 in Figure 3.3b), and conduct distance computations

only for objects thereafter (Lines 17–24).

Algorithm 3.3 Search and Cracking with Caching

```
1: procedure SEARCH-AND-CRACK-CACHING(data array O, query pt q, bound ε, node v, result R, threshold θ)
2:     if v.q is null then
3:         if v.size ≤ θ then                                              ▷ fixed leaf node
4:             qDist = distance(v,q)
5:             p(v) = parent of v
6:             if v.isLeftChild() then
7:                 if qDist > ε then                                       ▷ Case L1
8:                     low = first o ∈ v, such that d(p(v), o) ≥ qDist − ε
9:                     high = last o ∈ v, such that d(p(v), o) ≤ qDist + ε
10:                    for o ∈ v from low to high do
11:                        if d(q, o) ≤ ε then
12:                            R = R ∪ {o}
13:                else                                                    ▷ Case L2
14:                    low = first o ∈ v, s. t. d(p(v), o) ≥ ε − qDist
15:                    for o ∈ v from start until low (excl.) do
16:                        R = R ∪ {o}
17:                    for o ∈ v from low until end do
18:                        if d(q, o) ≤ ε then
19:                            R = R ∪ {o}
20:            else                                                        ▷ v is right child of p(v)
21:                if qDist ≤ ε + p(v).ε then                              ▷ Case R2
22:                    low = first o ∈ v
23:                else                                                    ▷ Case R1
24:                    low = first o ∈ v, such that d(p(v), o) ≥ qDist − ε
25:                high = last o ∈ v, such that d(p(v), o) ≤ qDist + ε
26:                for o ∈ v from low to high do
27:                    if d(q, o) ≤ ε then
28:                        R = R ∪ {o}
29:        else                                                            ▷ non-fixed leaf
30:            Lines 3–4 of Algorithm 3.1
31:            if v.left.size ≤ θ then                                     ▷ v.left is fixed
32:                qsort(v.left)
33:            if v.right.size ≤ θ then                                    ▷ v.right is fixed
34:                qsort(v.right)
35:        Lines 6–15 of Algorithm 3.1
```



(a) case L1

(b) case L2

(c) case R1

(d) case R2

Figure 3.3: Use of cached distances at AV-tree leaves

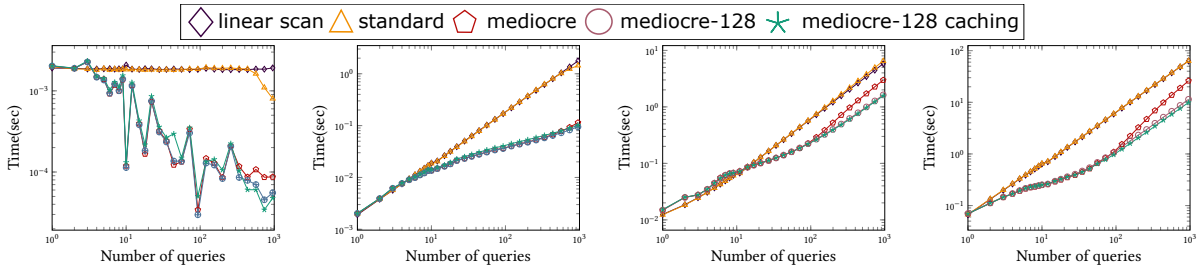
In **Case R1**, the ranges of query q and $p(v)$ are disjoint, i.e., $d(q, p(v)) > p(v).\epsilon + \epsilon$. Then, objects o outside the query range of $p(v)$ with $d(p(v), o) < d(q, p(v)) - \epsilon$ or $d(p(v), o) > d(q, p(v)) + \epsilon$ (e.g., p_1 and p_3 in Figure 3.3c) cannot be query results; thus, as in Case L1, we find, by binary search, the range of candidate query results and compute distances only for those. **Case R2** applies when $d(q, p(v)) \leq p(v).\epsilon + \epsilon$ and is similar to Case R1, except that now there are no objects o with $d(p(v), o) < d(q, p(v)) - \epsilon$ outside the query range $(p(v), p(v).\epsilon)$, hence a single binary search suffices.

In addition, when possible, we avoid binary search to compute the range of positions in which to scan objects and compute distances. For example, in Case L1, if $p(v).\epsilon \leq d(q, p(v))$, then there are no objects within the query range of $p(v)$ with $d(p(v), o) > d(q, p(v)) + \epsilon$, hence $high$ is the last position of v. Similarly, in Case L2, if $d(q, p(v)) + p(v).\epsilon \leq \epsilon$, then all objects within the query range of $p(v)$ are query results, hence

we need not do any comparisons. We also exploit the fact that, as objects in fixed leaves are sorted, the first and the last object in v provide lower and upper bounds to $d(p(v), o)$, respectively. Thus, in Cases R1 and R2, if the last cached distance $dLast$ is $dLast \leq d(q, p(v)) + \epsilon$, then we set $high$ to the last object position in v and eschew binary search.

3.2.4 Cost Analysis

Here, we analyze the cost of the AV-tree index with all enhancements. Assuming that AV-tree leaves of size no larger than θ are not cracked, we expect the tree to reach its maximum size after a large number of queries, whereupon no more cracks are performed. In this state, each leaf has $\theta/2$ objects on average, so the expected number of leaves is $\frac{2n}{\theta}$, where n is the number of objects in O. Since the AV-tree is a binary tree, the expected number of nodes is $\frac{4n}{\theta} - 1$, hence the index space complexity is $O(n/\theta)$. The worst-case cost of query processing is $O(n)$, accessing all leaves and data objects and computing their distances to q. So is the cost of the first query over an uncracked array. However, after a large number of queries, we expect the cost per query to drop and to converge to that of using a fully built VP-tree [53]. Lastly, the space requirements of distance caching are $O(n)$, i.e., one scalar per object, whereas those for storing the D-dimensional data array are $O(Dn)$.



(a) Time per query (MNIST50) (b) Cumulative time (MNIST50) (c) Cumulative time (Synthetic) (d) Cumulative time (Words)

Figure 3.4: AV-tree versions, 100 selectivity range workload.

3.3 Experimental Evaluation

We evaluate AV-tree against the following competitors:

- **Linear scan** computes distances $d(q, o)$ for all $o \in O$ and does not perform any data array re-organization or indexing.

- **SimplePivot** is described in Chapter 2. After experimental tuning, we opted to set the number of pivots m to 5, which yields the best performance; this parameter value selection is consistent with the experimental setup in [41].

- **MVP-tree** [54], described in Chapter 2. We used the same implementation[2] as in [41]. MVP-tree has two parameters, bucket size (equivalent to the threshold θ in AV-tree) and arity (i.e., number of children per node). We experimentally found that the MVP-Tree performs best with bucket size 64 and arity 5.

- **AKD-tree** [18] is the state-of-the-art adaptive index for multi-dimensional points; we used the authors' implementation[3]. To prevent excessive tree growth, we select 128 as the size threshold after experimentally assessing various values. The original implementation handles rectangular queries, i.e., the L_{max} distance metric. To adapt it to the L_2 distance metric, we first perform an L_{max}-query for the surrounding tangent box using the given q and ϵ, and then filter false positives by a L_2-based linear scan.

- **SAT** is an implementation[2] of the Spatial Approximation Tree [52] (see Chapter 2), which has no construction parameters.

Table 3.1: Datasets used in experiments

Dataset	Cardinality	dimensionality	distance	size (MB)
MNIST	70k	5, 20, **50**, 100	L_1, $\boldsymbol{L_2}$	3–55
Words	650k	2-33	edit distance	8
Synthetic	50k, **100k**, 200k, 500k	100	L_1, $\boldsymbol{L_2}$	20–450

3.3.1 Experimental Settings

Environment. We compiled all codes in g++ 9.4.0 with flags `-O3`, `-mavx`, and `-march=native` and ran experiments on a 32GB Ubuntu 20.04.3 LTS machine with Intel Core i9-10900K CPU @3.70GHz.

[2]https://github.com/kaarinita/metricSpaces

[3]https://github.com/pdet/MultidimensionalAdaptiveIndexing

Datasets. We use two publicly available real datasets and synthetically generated high-dimensional vectors. Table 5.1 summarizes statistics about the data with the default values of parameters and distance metrics shown in **boldface**. We provide more details below.

- **MNIST**[4] is a database of 70K handwritten digits [87]. Each digit is stored as a grayscale image with a size of 28x28 pixels. MNIST has been used in numerous similarity search studies (e.g., [88, 89, 90]). We use the UMAP [91] dimensionality reduction method to create various vector representations of the data with D in 5–100.

- **Words** is a database of 650K proper nouns, acronyms, and compound words, taken from the Moby project[5], with lengths varying from 2 to 33 characters. On Words, the query goal is to find words that are similar to a given query string by edit distance.

- **Synthetic** are generated datasets of 10 non-overlapping, equally sized clusters comprising 20K–500K points in 100 dimensions, generated as isotropic Gaussian blobs by make_blobs function of Python's sklearn [92] with a standard deviation of 0.5.

Queries. We ran workloads of range and kNN queries. In line with previous work [1, 18], our query workload consists of 1000 randomly sampled query points from the target dataset.[6] For range queries, we tuned ϵ to ensure that queries return the desired number of results. Query selectivity ranges from 20 to 1000 (default 100). For kNN queries, k ranges in $\{5, 20, 50, 100\}$ (default $k = 20$).

Cost measures. We evaluate all methods by their (i) cost per query and (ii) cumulative cost, as the query workload progresses; we average results over 5 runs. As SimplePivot, MVP-tree, and SAT are built in advance, we add their construction cost to the cumulative cost prior to the first query. Linear scan, AV-tree, and AKD-tree do not bear preprocessing costs.

[4] http://yann.lecun.com/exdb/mnist/

[5] https://en.wikipedia.org/wiki/Moby_Project

[6] In most real applications in metric spaces, queries are not ad hoc, but follow the data distribution.

3.3.2 Enhancements and parameter tuning

Effect of AV-tree enhancements

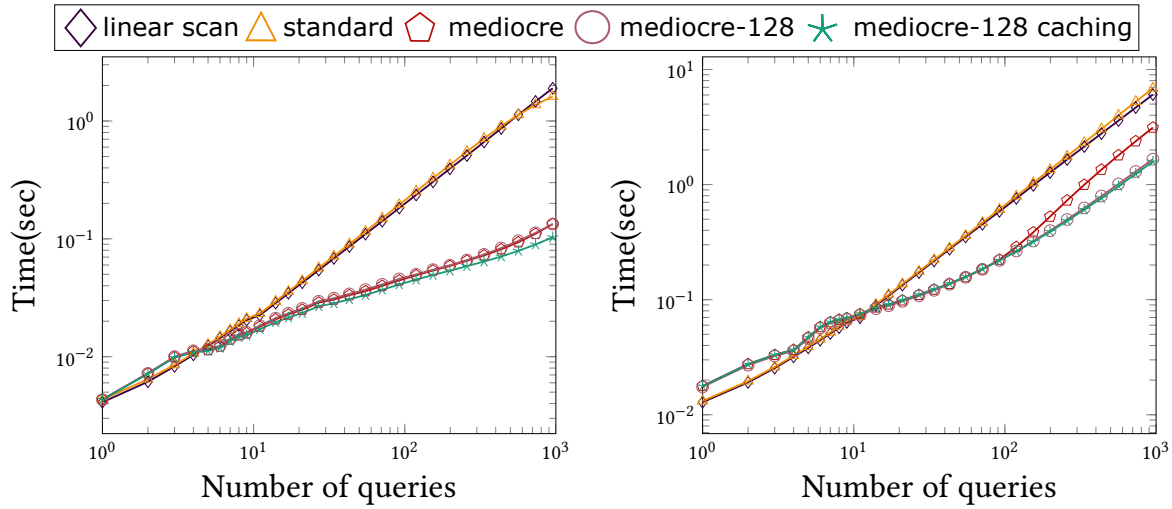
First, we evaluate different AV-tree versions with 1000 queries on the 50D MNIST dataset, including the performance of linear scan for reference. We compare the basic version of AV-tree, which uses standard cracking without any enhancements (labeled 'standard') to (i) its variant using mediocre cracking (Section 3.2.3, 'mediocre'); (ii) a variant using mediocre cracking and threshold $\theta = 128$ (Section 3.2.3, 'mediocre-128'); and (iii) a variant that applies all enhancements including caching (Section 3.2.3, 'mediocre-128 caching').

Figures 3.4a and 3.4b show the per-query and cumulative costs, respectively, of all AV-tree variants on MNIST, while Figures 3.4c and 3.4d show their cumulative costs on Sythetic and Words. Notably, both mediocre cracking and thresholding boost performance, with the effect of thresholding being smaller on MNIST. Mediocre cracking creates a balanced tree, as each crack splits a leaf in two partitions of roughly equal size, while thresholding avoids building an excessively tall tree, which would be detrimental to performance, as its traversal does not pay off compared to the achieved savings. On the other hand, caching pays off in cases where distance computation is expensive, e.g., on the Words data, where we use edit distance.

Table 3.2: AV-tree versions, MNIST50, post 1k range queries

	cum. time	cum. distance comp.	#nodes
Linear Scan	1.8548	70000	-
Standard	1.4831	38818.553	11983
Mediocre	0.1215	1602.105	85809
Mediocre-128	0.1019	1735.236	1843
Mediocre-128 caching	0.1002	1555.677	1843

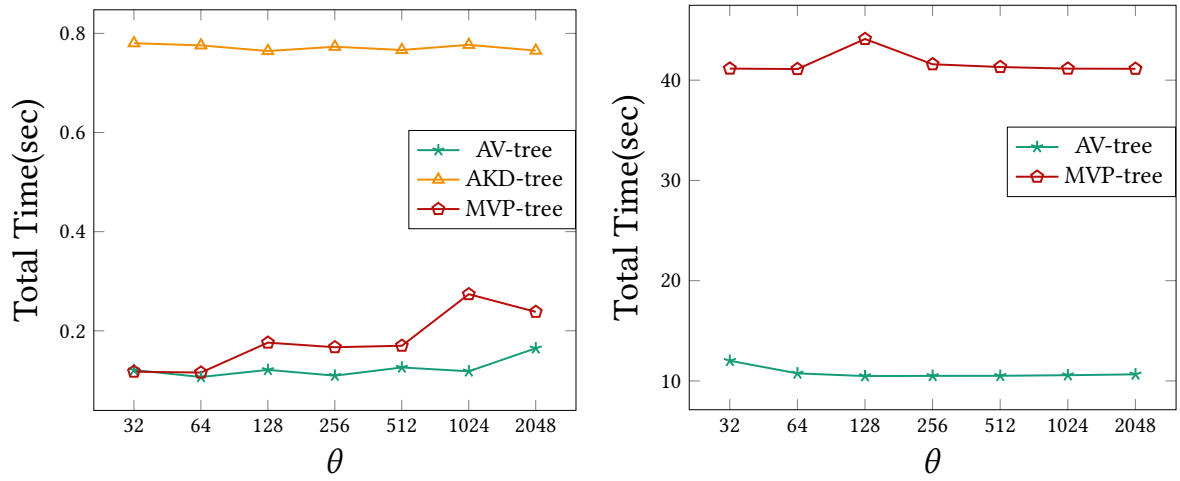
Table 3.2 shows the cumulative costs, distance computations and number of AV-tree nodes after 1000 range queries on the 50-dimensional MNIST. The fully optimized AV-tree surpasses all other versions in all respects. While on these data Mediocre without threshold performs similarly to Mediocre-128, it incurs a significant space overhead by building an AV-tree even bigger than Standard. The same also holds for all other datasets; we omit the corresponding tables in the interest of space.



(a) MNIST50 (b) Synthetic

Figure 3.5: L_1 distance, 100-selectivity range workload

Figure 3.5 shows the cumulative cost of AV-tree variants on MNIST and Synthetic using L_1 distance instead of L_2 (Euclidean). Note that the performance difference when using L_1 is insignificant. Henceforward, we adopt all enhancements in the AV-tree and use L_2 as a distance measure on MNIST and Synthetic.



(a) MNIST50 (b) Words

Figure 3.6: Parameter Tuning, 100 selectivity range workload

Parameter setting

We tuned the threshold parameter on AV-tree, AKD-tree, and MVP-tree on MNIST50 and Words. Figure 3.6 plots the total cost for an 1K-query workload vs. different

threshold values. As the plot shows, the optimal threshold values for AV-tree, AKD-tree, and MVP-tree, are 128, 128, and 64 respectively.

3.3.3 Comparative study

MNIST

Next, we try the fully enhanced AV-tree vs. the competitors listed in Section 3.3 with range and kNN queries on MNIST.

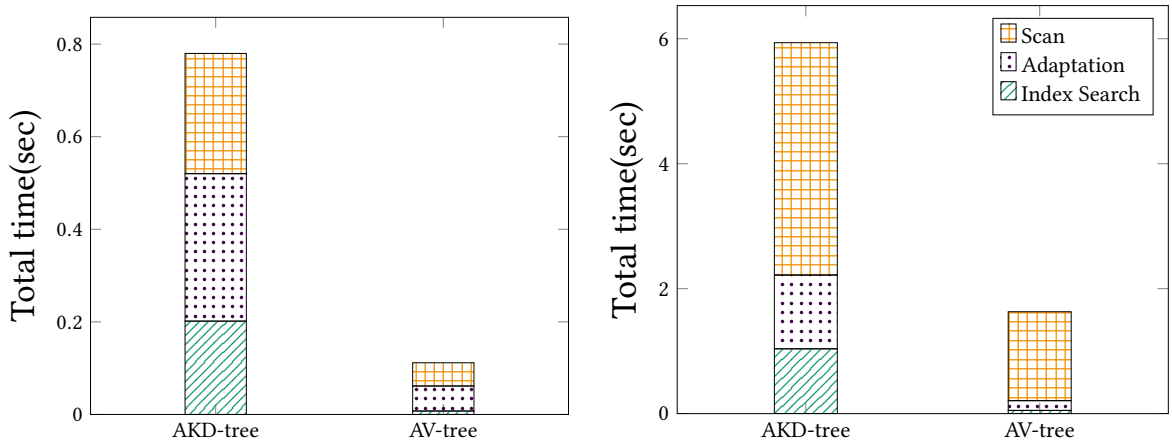
Dimensionality Figure 3.8 shows the per-query and cumulative cost, as the query workload (selectivity $s = 100$) progresses, on MNIST datasets of varying dimensionality. AV-tree exhibits the ideal behavior of an adaptive index: its per-query cost gradually drops and reaches that of the MVP-tree. Its cumulative cost outpaces all competitors and eventually matches the MVP-tree. This progression is slower on lower dimensionality; on higher dimensionality, the two lines meet after around 100 queries. The AKD-tree performs competitively to the AV-tree only on very low dimensionality (D=5), where hyperplane-based partitioning works satisfactorily. Until it reaches the size threshold, the AKD-tree creates $2D$ new levels per crack, leading to an exorbitantly tall tree that is expensive to traverse, hence its disadvantage on higher dimensionality. SimplePivot is inferior to MVP-tree and SAT, especially when D is small. These results are consistent with the findings in [41]. MVP-tree has lower per-query cost than SAT in data of medium dimensionality, but the two costs are similar in high-dimensional spaces. Still, SAT incurs a very high start-up (i.e., construction) cost compared to MVP-tree.

Figure 3.9 repeats the experiment with kNN queries, setting k to 20. We excluded the AKD-tree from the comparison, as it does not support kNN queries. Our findings reaffirm those for range queries, as the data are reorganized (i.e., cracked) similarly in both cases, leading to a good data structure, while the use of the two priority queues in the AV-tree prevents redundant search.

Selectivity Figure 3.10 juxtaposes all methods on workloads of varying selectivity s; their relative performance is largely unaffected by selectivity, with the discernible exception of the AKD-tree, which is sensitive to large s due to the expensive L_2-filtering; as s grows, the items to be scanned increase. Cost is largely unaffected by k in kNN queries, as Figure 3.11 shows. The AV-tree is robust to selectivity, as cracking

is insensitive to the number of query results and thanks to the data structures it uses to manage kNN query results.

Cost breakdown Figure 3.7 breaks down the total runtime of the default range workloads for the AV-tree and AKD-tree on the default MNIST50 and Synthetic datasets. Total time comprises the costs for: (i) searching the index for relevant partitions (Index Search); (ii) index restructuring, i.e., creating new nodes and swapping (Adaptation); and (iii) scanning data objects in fixed leaves that are not being cracked further (Scan) — in the AKD-tree, Scan includes the time for L_2 filtering. All costs are higher for the AKD-tree: (i) index-search cost due to the ineffectiveness of hyperplane-based partitions and the larger index size, (ii) adaptation cost due to generating more (hyperplane-based) partitions than the AV-tree, (iii) scan cost due to refining spherical range queries.



(a) MNIST50 (b) Synthetic

Figure 3.7: Cost Breakdown

Words

We next compare all methods for range and kNN query workloads on the Words data. We omit the AKD-tree, as it does not support non-vector data and non-L_p distance measures. First, we create range query workloads by picking 1000 random words of fixed length (4 to 10), set $\epsilon = 2$, and measure the cumulative cost of all methods. As Figure 3.12 shows, AV-tree outperforms all competitors, and fares better on smaller query word lengths. With longer words, the curse of dimensionality comes into play and all index-based methods acquire costs similar to linear scan; yet even then, AV-tree outpaces SimplePivot and matches the MVP-tree.

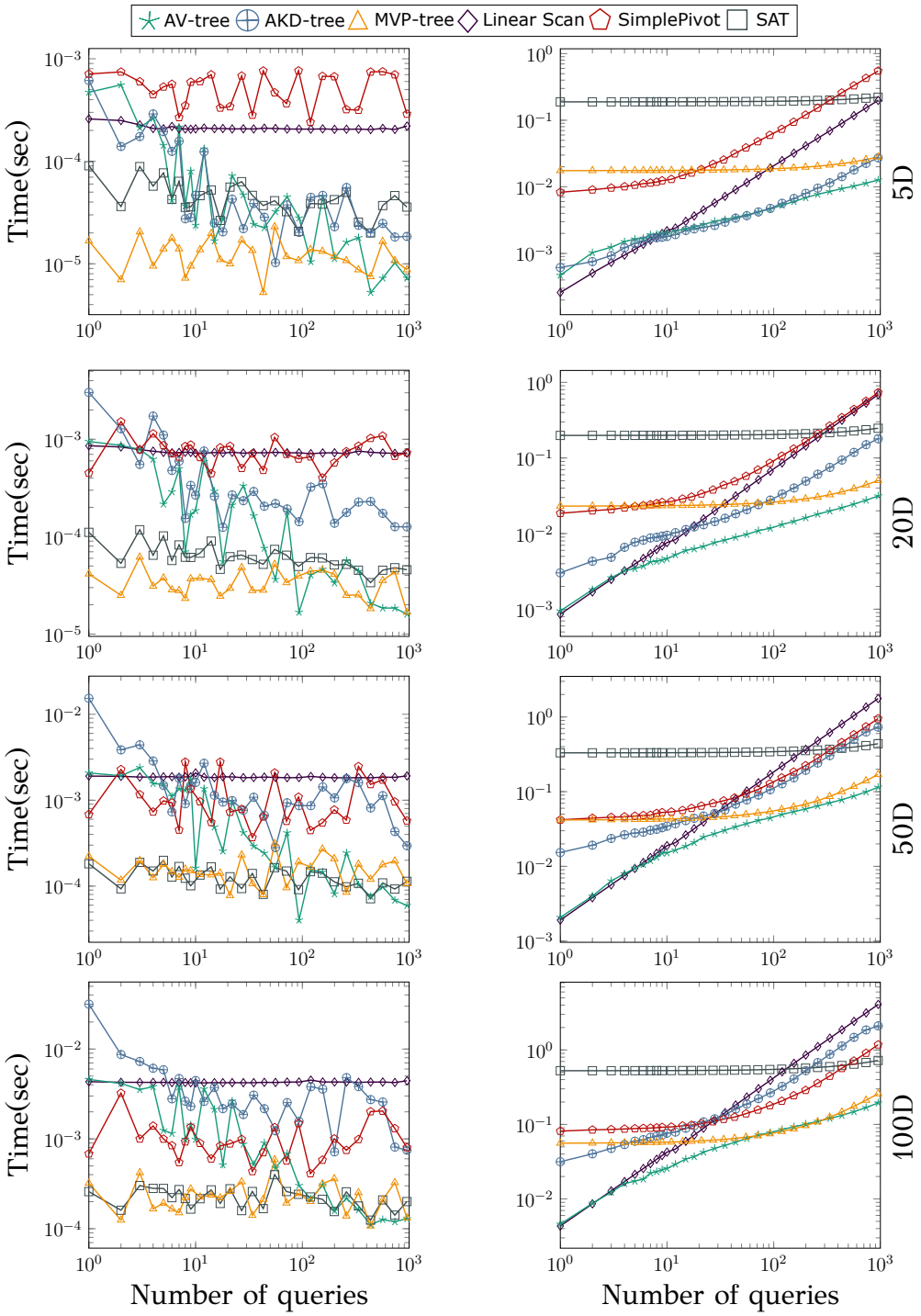


Figure 3.8: Effect of dimensionality, MNIST data, 100-selectivity range workload, per query (left) and cumulative time (right).

Figure 3.13 juxtaposes all methods the same workload of length-6 queries, tuning the values of ϵ, i.e., varying selectivity. With more selective queries (lower ϵ), index-based methods outperform linear scan, and the AV-tree gains an advantage. However, indices are less effective with less selective queries ($\epsilon = 4$), thus the AV-tree advantage

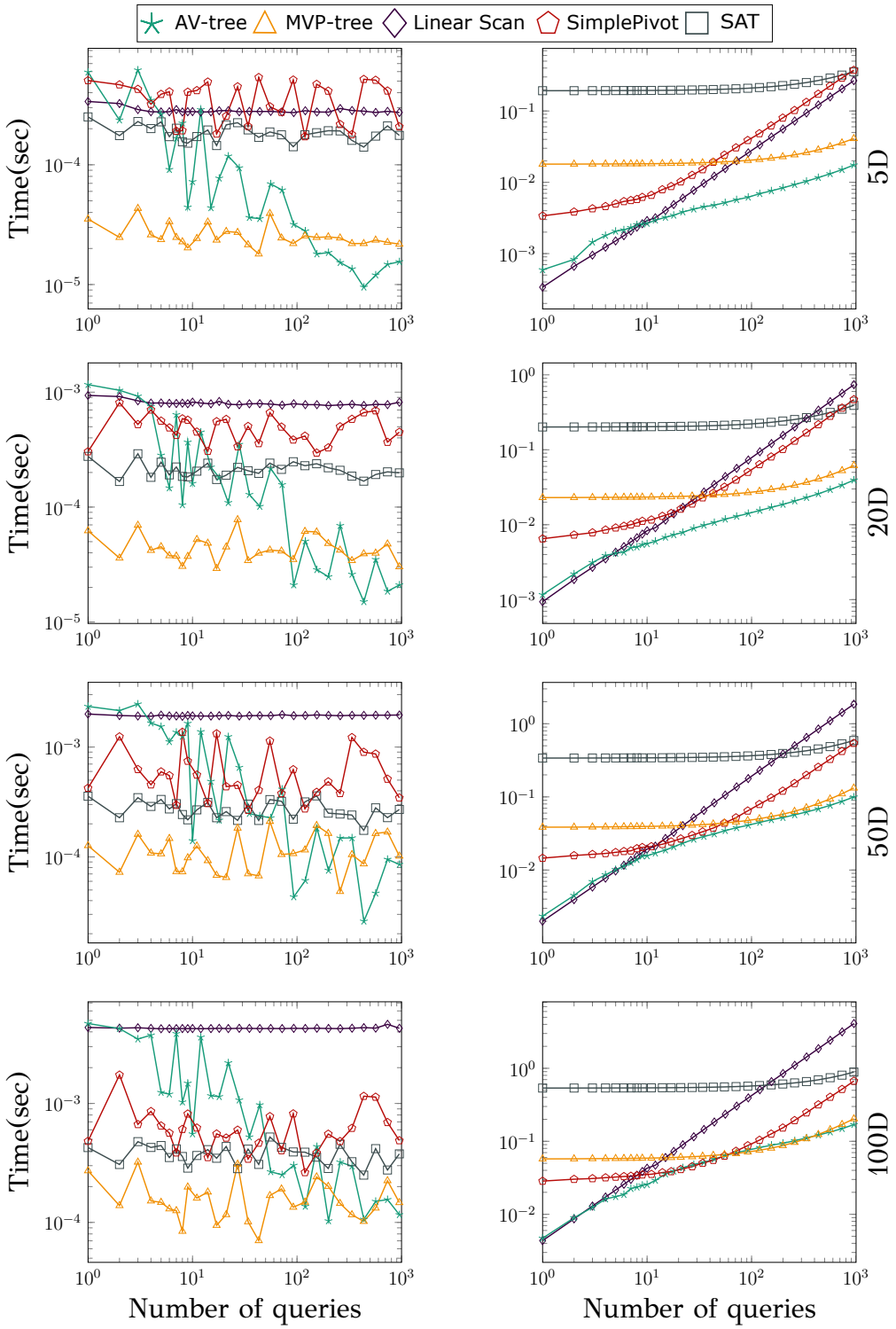


Figure 3.9: Effect of dimensionality, MNIST data, 20NN workload, per query (left) and cumulative time (right).

diminishes. Lastly, Figure 3.14 shows the performance of AV-tree on kNN queries vs. the value of k. The results resemble those for range queries of varying selectivity.

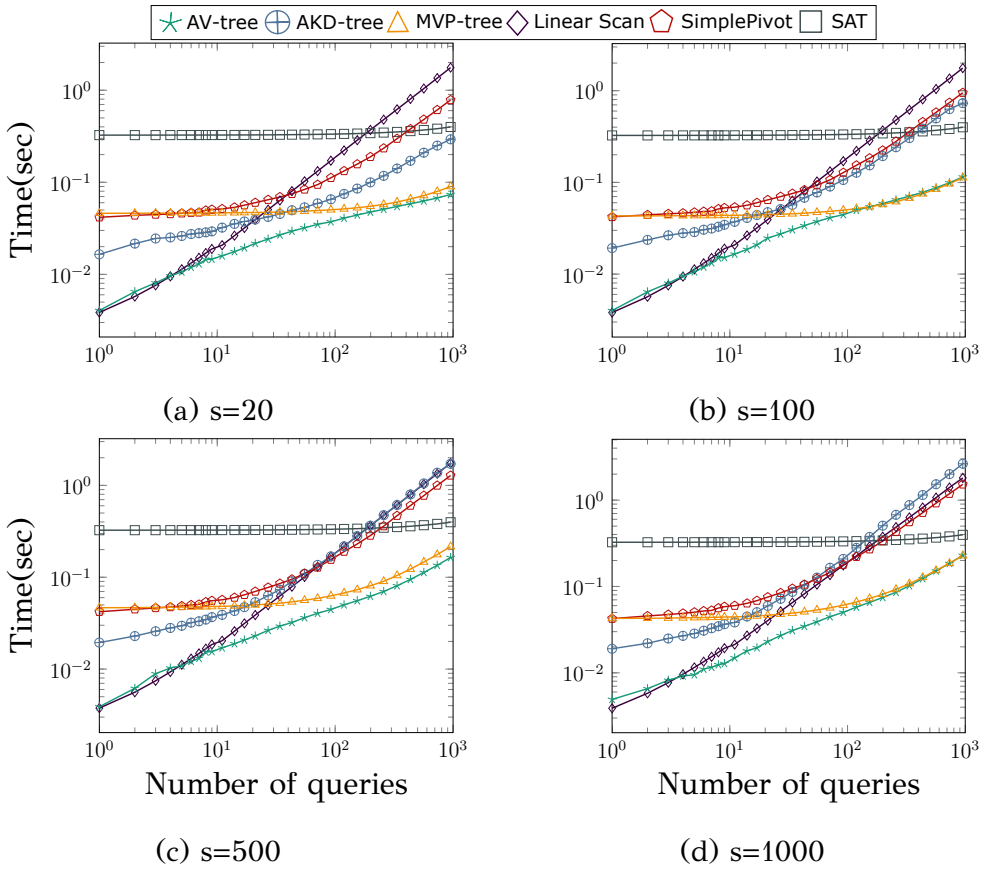


Figure 3.10: Effect of selectivity, MNIST50 data, range workload, cumulative time.

Overall, AV-tree presents an ideal behavior on the Words dataset, as its cumulative cost is consistently below that of all other methods, with the difference being more striking in the first few hundreds of queries.

Synthetic data

We now compare the performance of all methods against synthetically generated datasets of different scale, generated as described in Section 3.3.1. Figure 3.15 shows cumulative costs on range query workloads. Noticeably, the superior performance of the AV-tree is insensitive to data scale; its cost is close to that of linear scan in the first few queries and matches that of MVP-tree after a few hundreds of queries, while linear scan remains too slow. AV-tree is equally robust to data scale on kNN query workloads, as Figure 3.16 shows.

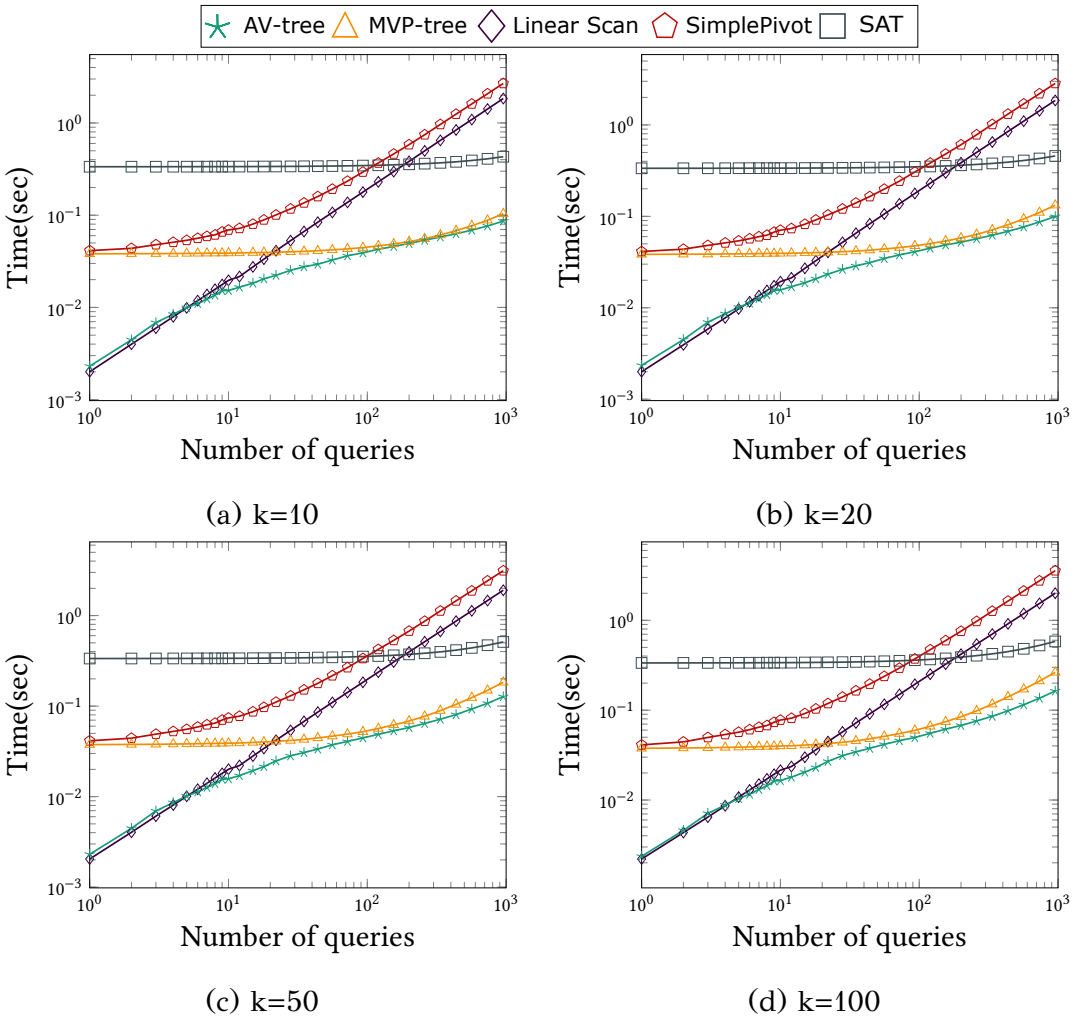


Figure 3.11: Effect of k, MNIST50 data, kNN workload, cumulative time.

Table 3.3: Index size (MB) after 1K range queries.

	AV-tree	AV-Tree (no cache)	AKD-tree	MVP-tree
MNIST	0.3989	0.1189	2.7989	0.2864
Words	3.2654	0.6654	-	2.7
Synthetic	0.5682	0.1682	2.6732	0.2909

3.3.4 Index Size

Lastly, we compare the *eventual* index sizes of AV-tree, AKD-tree, and MVP-tree, after the default workload of 1K range queries of selectivity 100. As Table 3.3 shows, AV-tree is lightweight, as it has size controlled by a threshold and caches at most one distance per object. Compared to the size of the corresponding datasets in Table 5.1, AV-tree occupies little space; this is yet another advantage of our method. If we eschew distance caching in AV-tree (2nd column), the index becomes even smaller

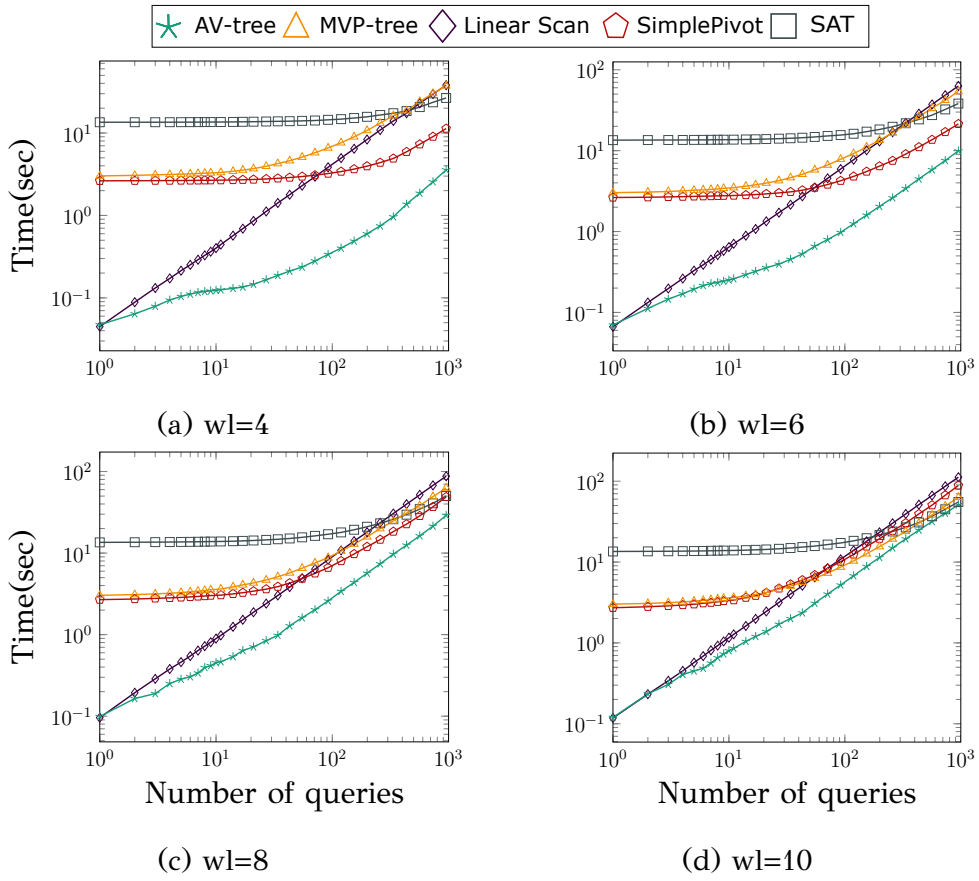


Figure 3.12: Effect of query length, Words data, edit distance $\epsilon = 2$, cumulative time.

than that of MVP-tree, at the price of a small overhead in the search performance.

3.4 Conclusions

We introduced the *adaptive vantage* tree (AV-tree), the *first*, to our knowledge, adaptive index tailored for high-dimensional metric spaces. In manner reminiscent of previously proposed adaptive indices for single columns [1, 17] and for a few attributes [19, 18], the AV-tree gracefully adapts to a query workload to progressively build a complete high-quality index. Nevertheless, unlike previous adaptive indexing methods, the AV-tree partitions the space around query centers into units defined by hyperspheres using *mediocre* distance bounds that naturally adapt to the data distribution, rather than into orthotopes (i.e., hyperrectangular units). Our experimental study on two real datasets of different natures, with diverse distance metrics, demonstrates that the AV-tree achieves low cumulative query cost compared to (i) iteratively

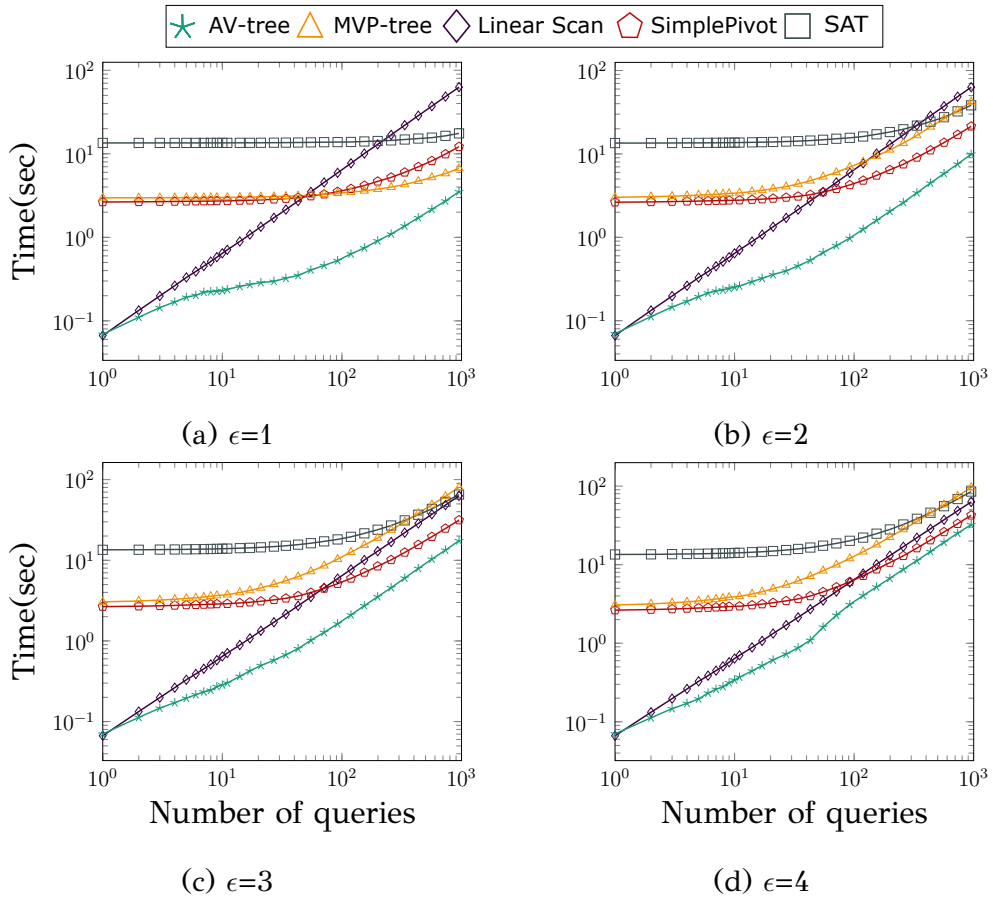


Figure 3.13: Effect of ϵ, Words data, 6-letter-word queries, range workload, cumulative time.

applying a linear scan; (ii) using a pre-built MVP-tree, the state-of-the-art index for *metric spaces*; and (iii) employing the AKD-tree, the state-of-the-art *adaptive* index for multidimensional data. In the future, we intend to investigate the performance of a multiway AV-tree (MAV-tree) that, unlike the current binary space partitioning, will divide the space around each query into layers based on several distance bounds. We also intend to examine alternative ways of measuring distance [93] and adaptive multidimensional synopses [94].

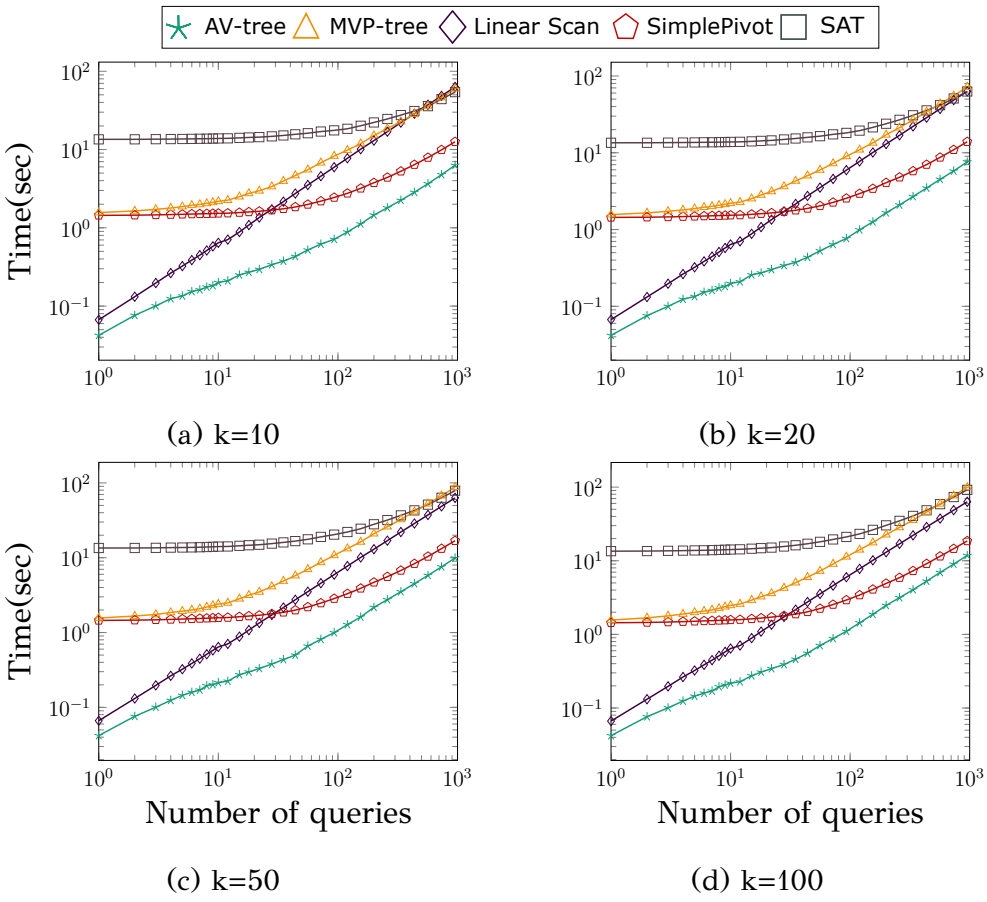


Figure 3.14: Effect of k, Words data, 6-letter-word kNN queries, cumulative time.

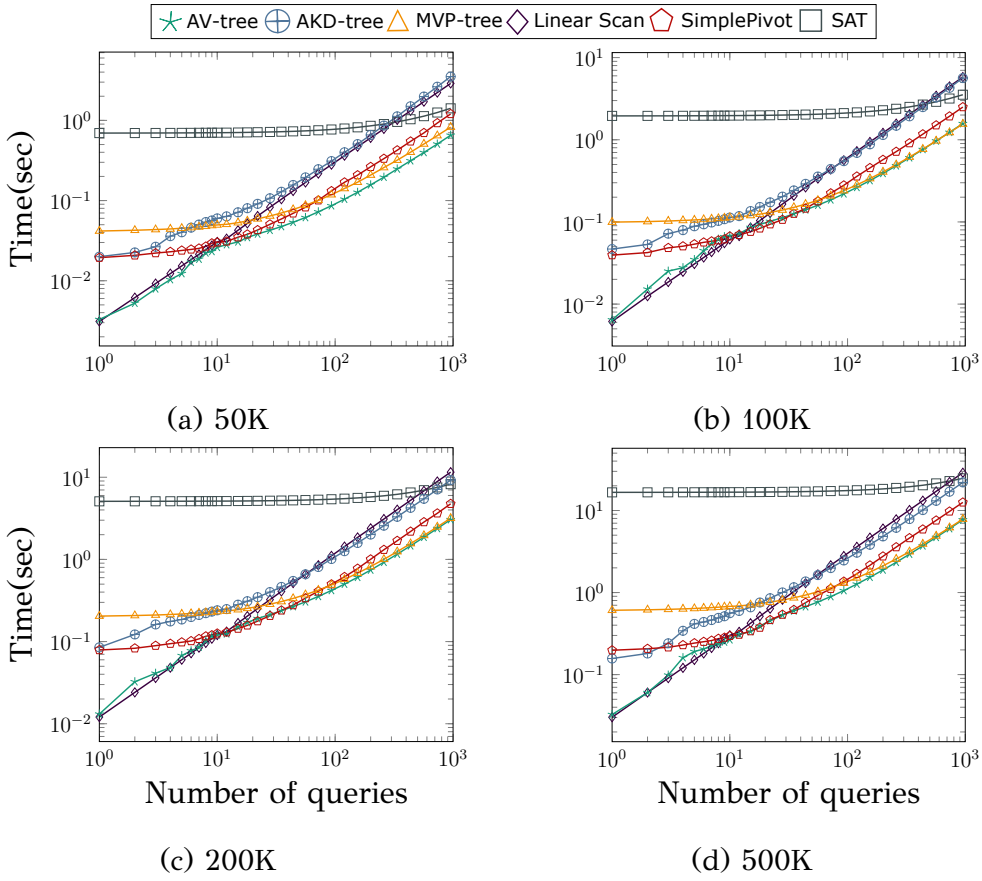


Figure 3.15: Effect of data size, Synthetic 100D data, 100-selectivity range workload, cumulative time.

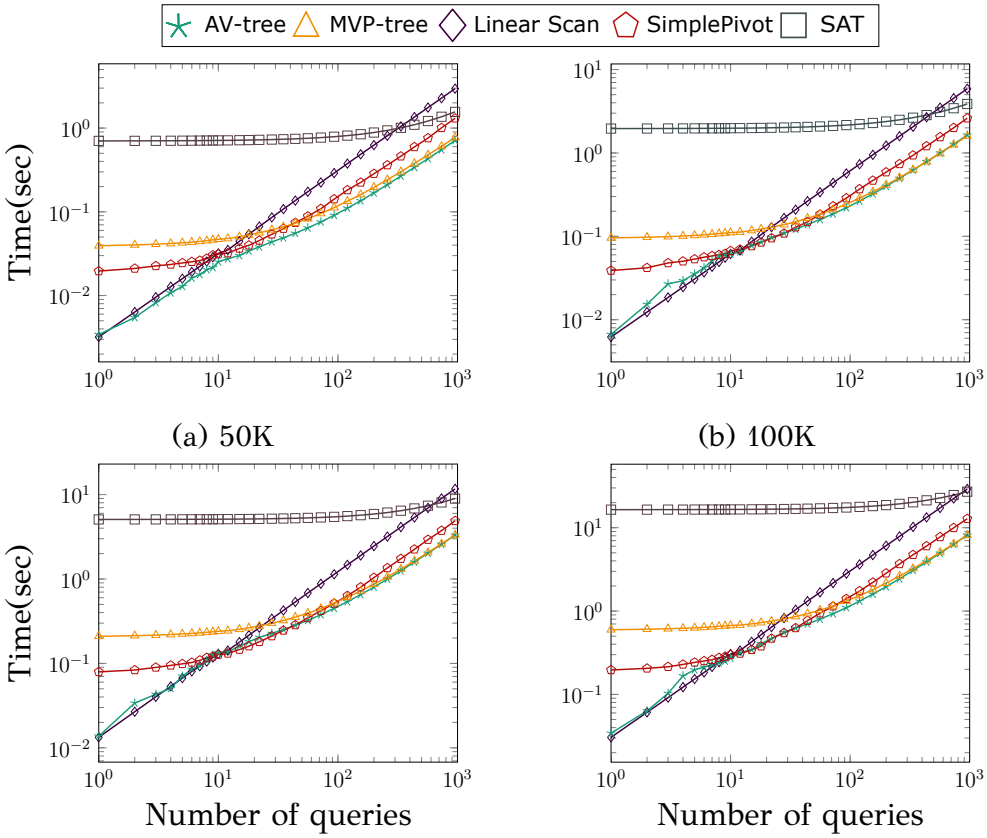


Figure 3.16: Effect of data size, Synthetic 100D data & 20NN workload, cumulative time.

Chapter 4

Benchmarking Adaptive Multidimensional Indices

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 indices 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. In the case of multidimensional range queries, various adaptive indexing techniques have been developed, each with distinctive strengths. Despite this significant body of work, there remains a gap in comparative studies that evaluate these methods on equal terms in a wide array 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.

Outline. Section 4.1 offers an overview of the methods compared in this study. Section 4.2 details the experimental setup, including the datasets, query workloads, and the methodology used for evaluation. Section 4.3 presents the experimental results and analysis. Finally, Section 4.4 summarizes our findings, discusses the results, and provides guidelines for selecting the most appropriate methods in various use cases.

4.1 Methods

Here, we describe the multidimensional (adaptive) indices that we compare experimentally. We focus on in-memory indexing of multidimensional points and hyper-rectangular ranges, which we call, for simplicity, boxes. Table 4.1 summarizes the compared methods and classifies them into three categories based on their adaptiveness. Pre-bulit, i.e, *non-adaptive* indices are fully constructed before query processing. The second class includes adaptive indices 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 [25] in the multidimensional space; this is not a purely adaptive index, as it first coarsely partitions the data and then indexes the partitions adaptively.

4.1.1 Non-adaptive indices

Non-adaptive indices are general-purpose index structures for multidimensional data, applicable to index multidimensional points and ranges. These include the R-tree [31], the quadtree [34], and a multidimensional uniform grid [39]. 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 [95]. 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 [39]. While they perform best on non-skewed data, they were shown to perform well for moderately skewed data [39]. R-tree partitions may overlap on each level of the hierarchy, while quadtree and grid partitions are spatially disjoint.

4.1.2 Adaptive indices

The *QUery-Aware Spatial Incremental Index* (QUASII) [19] is the first proposed multi-dimensional 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. As it is designed for multidimensional ranges (i.e., spatial objects), QUASII adjusts partition boundaries to contain the query results. 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. Using QUASII for point data is straightforward by skipping the query adjustment.

The adaptive KD-tree (AKD) [28, 18] repetitively cracks a leaf node of the binary search 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 to build a (binary) KD-tree.

The *Progressive KD-Tree* (PKD) and its extension, the Greedy Progressive KD-Tree (GPKD) [28], 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) [21] progressively constructs an in-memory R-tree [31], 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 of spatial objects), AIR is also applicable on point data. Its performance is affected by a pathological scenario where queries come in a sequential spatial order. To mitigate these effects, AIR applies

Table 4.1: Classification of tested methods

	Method name	points	boxes	replication (boxes)	tree	tree balance	overlapping partitions
Pre-built	R-tree [31]	✓	✓	X	✓	✓	✓
	Quadtree [34]	✓	✓	✓	✓	X	X
	grid [39]	✓	✓	✓	X	–	X
Adaptive	QUASII [19]	✓	✓	X	✓	✓	X
	AKD [28, 18]	✓	X	X	✓	X	X
	AIR [21]	✓	✓	X	✓	✓	✓
	AV-tree [22]	✓	X	X	✓	X	✓
	AAKD (combines [18] and [21])	✓	X	X	✓	X	X
Hybrid	CGI (applies CGI [25] to kD spaces)	✓	X	X	X	X	X

stochastic cracking on the largest piece resulting from a query.

The AV-tree [22] is designed for similarity queries in high-dimensional spaces. We apply it on rectangular range queries as follows. 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 $q.b^d = (q^d.high - q^d.low)/2$. 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:

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

QUASII and AIR can handle points and rectangles, whereas AKD and AV-tree are designed for point data. AKD is applicable to rectangles by treating each data rectangle as a point formed by its lower bound at each dimension and extending the query range to the maximum object extent in each dimension [96].

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. [21] 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.

4.1.3 Hybrid indexing

The *Coarse Granular Index* (CGI) [25] 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 4.1 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 partitoin sparsity, grow exponentially with dimensionality.

Using an irregular grid.

While CGI has been proven to be very effective for uniform 1D data [26], it has not been tested for non-uniform data, where it might falter due to load imbalance between partitions. On skewed multidimensional data distributions, in addition to testing CGI with a regular grid, we also test an alternative approach that defines *irregular* domain partitions in each dimension, following the data distribution. As finding an optimal irregular grid is costly, we collect a data sample to find the dividers in each dimension

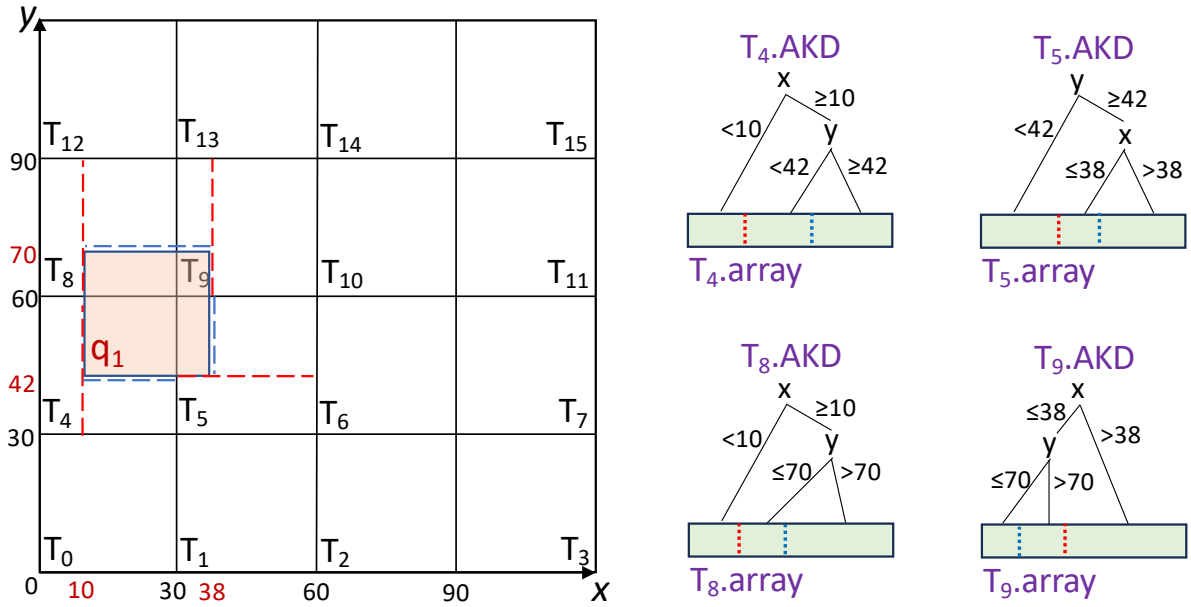


Figure 4.1: Multidimensional GCI using AKD

that partition the domain into cells of approximately equal cardinality. We sort the samples data on the x-axis and identify the values (i.e., quantiles) that divide the sample into equal parts. We repeat this process on the y-axis to generate an irregular grid based on the data distribution. The resulting grid balances cell cardinality, aiming to more efficient indexing and query processing on non-uniform data distributions.

4.2 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 4.2.1), query workloads (Section 4.2.2), and performance measures (Section 4.2.3). We also conduct tuning experiments (Section 4.2.4) that suggest the most robust parameter values for the compared methods.

4.2.1 Datasets

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

Table 4.2: 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

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 [39, 21]. 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's 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.

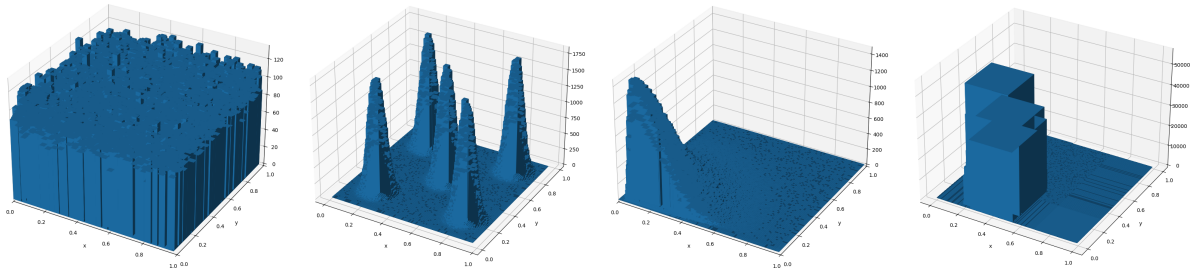
- **Skewed data.** We generate random values by the *skewnorm* function from the SciPy library in Python, which produces data that follow a skew-normal distribution, which extends the normal distribution to incorporate non-zero skewness, allowing for asymmetric data shapes. We set the skewness parameter 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 $[0.0001, 00.2]$ 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 $g(u) = 3^{-3u}$.

Real data

As real data, we utilize the publicly available **ROADS** dataset which features the shape of roads in the US [97]. 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

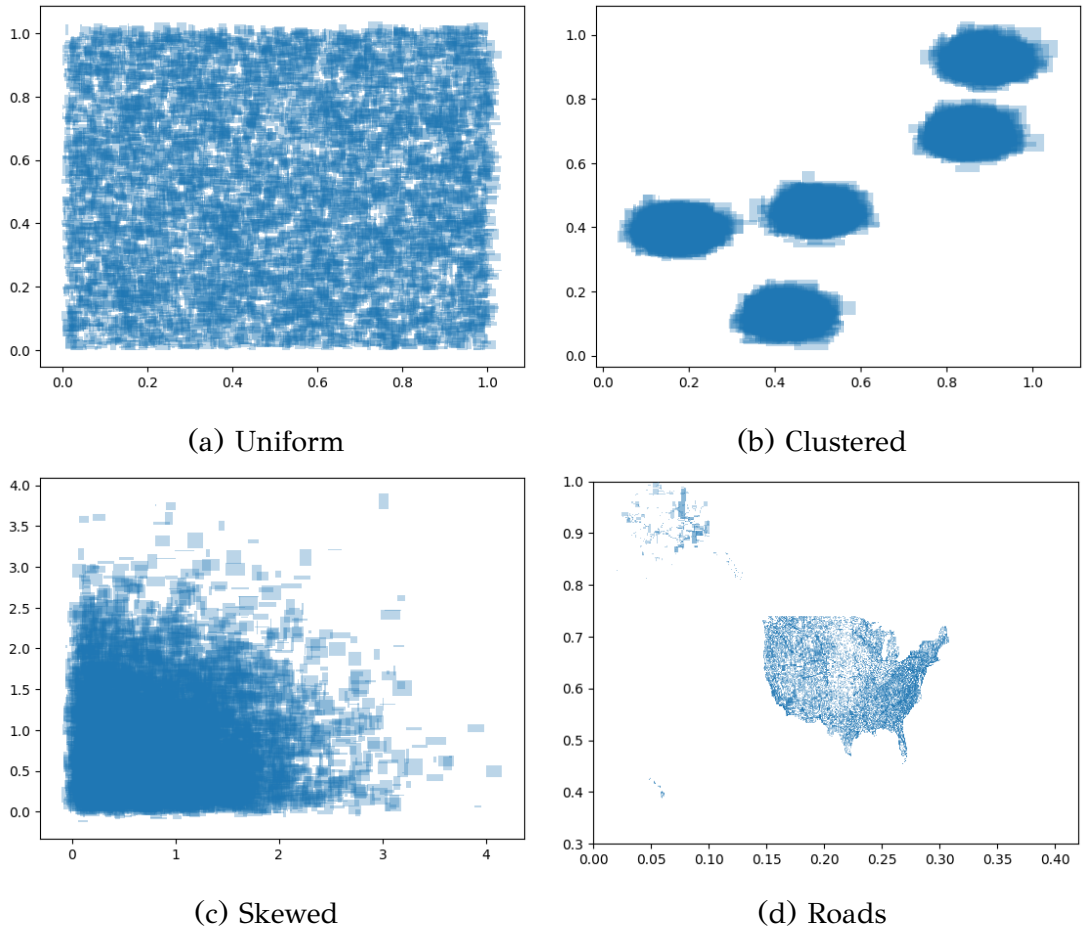
Figure 4.2: Distribution of point datasets

Figures 4.2 and 4.3 visualize the distribution of our data, both synthetically generated and real, points and ranges. The distribution 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.

4.2.2 Workloads

Figure 4.4 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 adap-

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

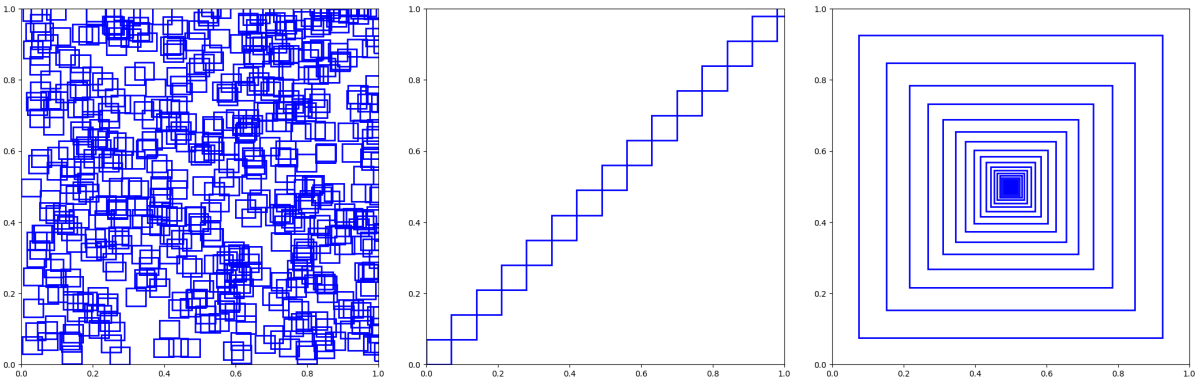


(a) Uniform (b) Clustered

(c) Skewed (d) Roads

Figure 4.3: Distribution of shape datasets

tive 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 4.4: Access pattern of synthetic workloads

4.2.3 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. We assess robustness to varying data distributions, query workloads, data dimensionality, object size and cardinality.

4.2.4 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.

Table 4.3: 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

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 4.3 and 4.4 show the total workload time. 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.

Table 4.4: 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

On the irregular 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 4.5 details the search for the best empirical values of these parameters.

Table 4.5: 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	

4.3 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 (§4.3.1).

2. Next, we thoroughly evaluate the top methods across various types of data (points and ranges), with different distributions of object locations (§4.3.2) and query access (§4.3.6).

3. Then, we assess how the distribution of object extents affects performance (§4.3.3).

4. We also examine the effect of dataset size (§4.3.4) and query selectivity (§4.3.5).

5. Finally, we examine the effect of dimensionality using high-dimensional point data (§4.3.7).

We plot the results in groups to maintain readable plots.

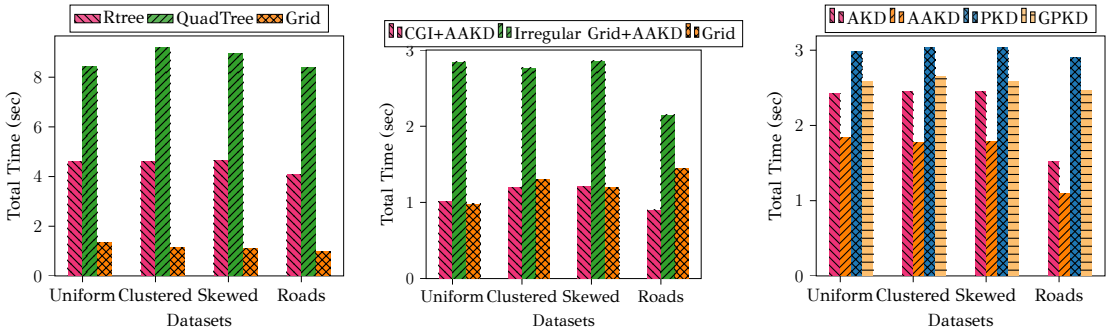
Since QUASII and AKD-based methods are not designed to handle data with spatial extent, we adjust them to achieve proper functioning with the *query window extension* technique [96] on data with spatial extent. We adjust the lower coordinates of the query window by the maximum object extent in the data, 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.

4.3.1 Method Selection

Static Indices

We refer to indices that are pre-built before evaluating any query as *static*. These structures remain unchanged throughout the query workload. Figures 4.5a and 4.6a illustrate the total workload time for index construction and evaluation of 10k queries, for three static 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 indices, such as the R-tree and the quadtree, take longer time to construct, which may pay off only after extremely long query workloads. For workloads intended for adaptive indices (such as 10K queries), 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.



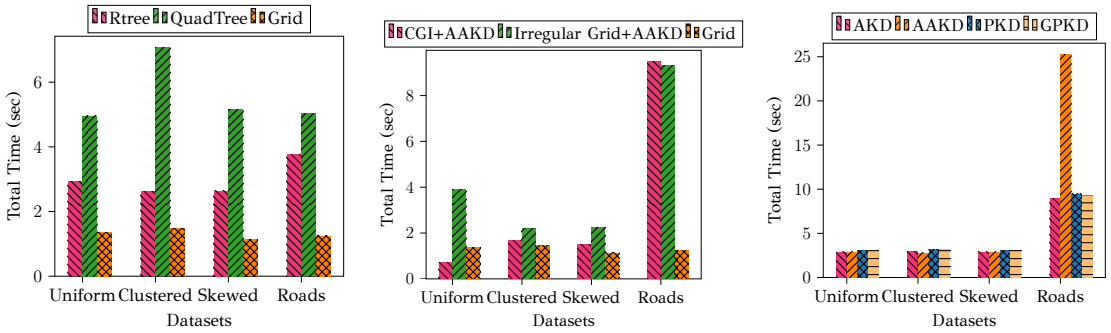
(a) Static (b) Grid-based (c) AKD-based

Figure 4.5: Comparison of indices in each category (cumulative time), point data

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 4.5b and 4.6b 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 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 the regular grid combined with the AAKD to representative the CGI index in subsequent experiments.



(a) Static (b) Grid-based (c) AKD-based

Figure 4.6: Comparison of indices in each category (cumulative time), shape data

AKD Indices

Three adaptive KD-tree variants are suggested in [18]: 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 4.5c and 4.6c 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 following, we use AAKD as the representative of AKD methodologies. However, since AKD is the original method, we have chosen to include it in the experiments as well.

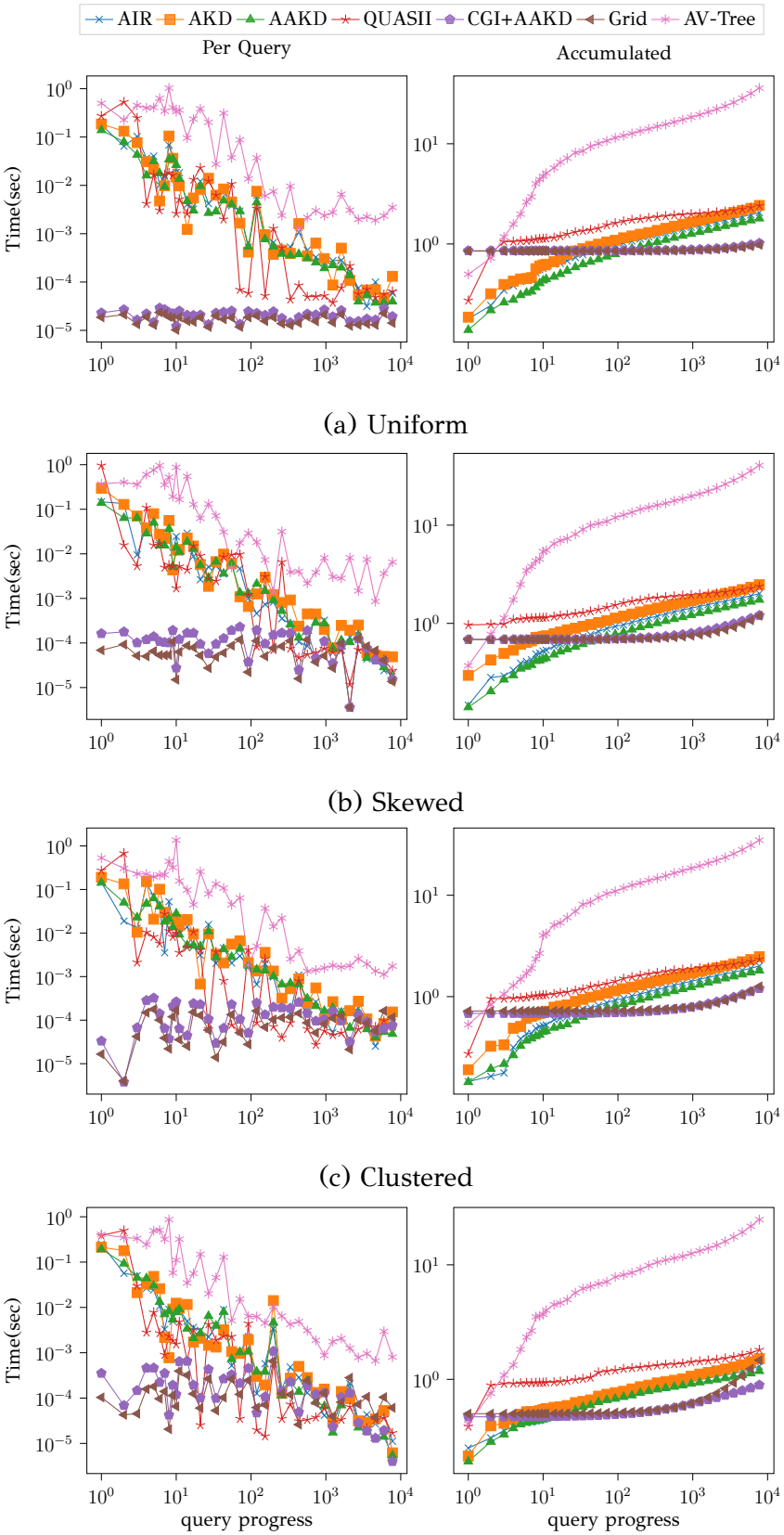
4.3.2 Effect of object location

Point Data

We commence our comparative study for static, adaptive, and hybrid indexing options with 2D point data. Figure 4.7 shows how the methods selected in Section 4.3.1, i.e., static grid, CGI+AAKD, and AAKD, perform alongside the adaptive indices: 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. We discuss experiments with range data in the following. These three indices 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 (§4.2.4), 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 4.2 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 rectangular range queries; as the index was



(a) Uniform

(b) Skewed

(c) Clustered

(d) Roads

Figure 4.7: Effect of data distribution on 2D point datasets.

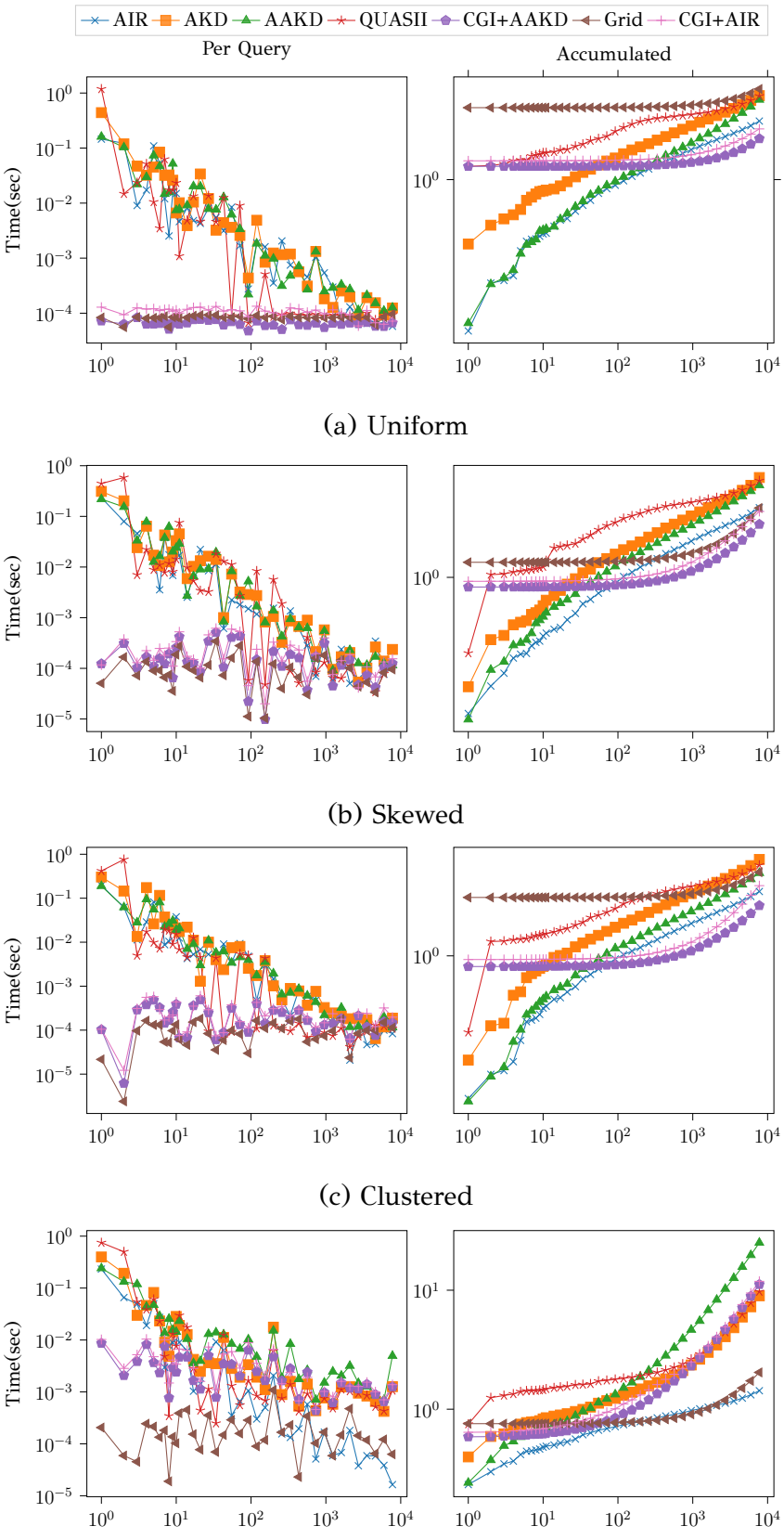
designed to accommodate L_2 distance queries, it creates suboptimal partitions when faced with L_{max} distance queries. And (ii) the low data dimensionality, which renders distance-based partitioning less appropriate. The combination of these two factors contributes to AV-tree's poor results. As its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

Shape Data

Figure 4.8 shows the per-query and cumulative workload time for the methods in Section 4.3.2 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 4.3.3.

The performance of AAKD, AKD, and QUASII has shifted up compared to the point experiments due to the query-window-extension handicap they have to endure to process shape datasets. This effect makes AAKD and AIR perform similarly for uniformly sized objects. 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 size of the initial grid in the CGI matches that of the static. As Figure 4.8 shows, the initial cost of creation is much higher when dealing with replication. Thus, we decided to go with the second option. This makes the likelihood of a better performance from this hybrid very low, but we have included it for completeness. This is also proven in the results that show the almost equivalent performance of the two CGI methods.

Finally, 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



(a) Uniform

(b) Skewed

(c) Clustered

(d) ROADS

Figure 4.8: Effect of data distribution on 2D shapes.

Table 4.6: 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]	[$2.05e^{-8}$, $1.39^{e}-9$]

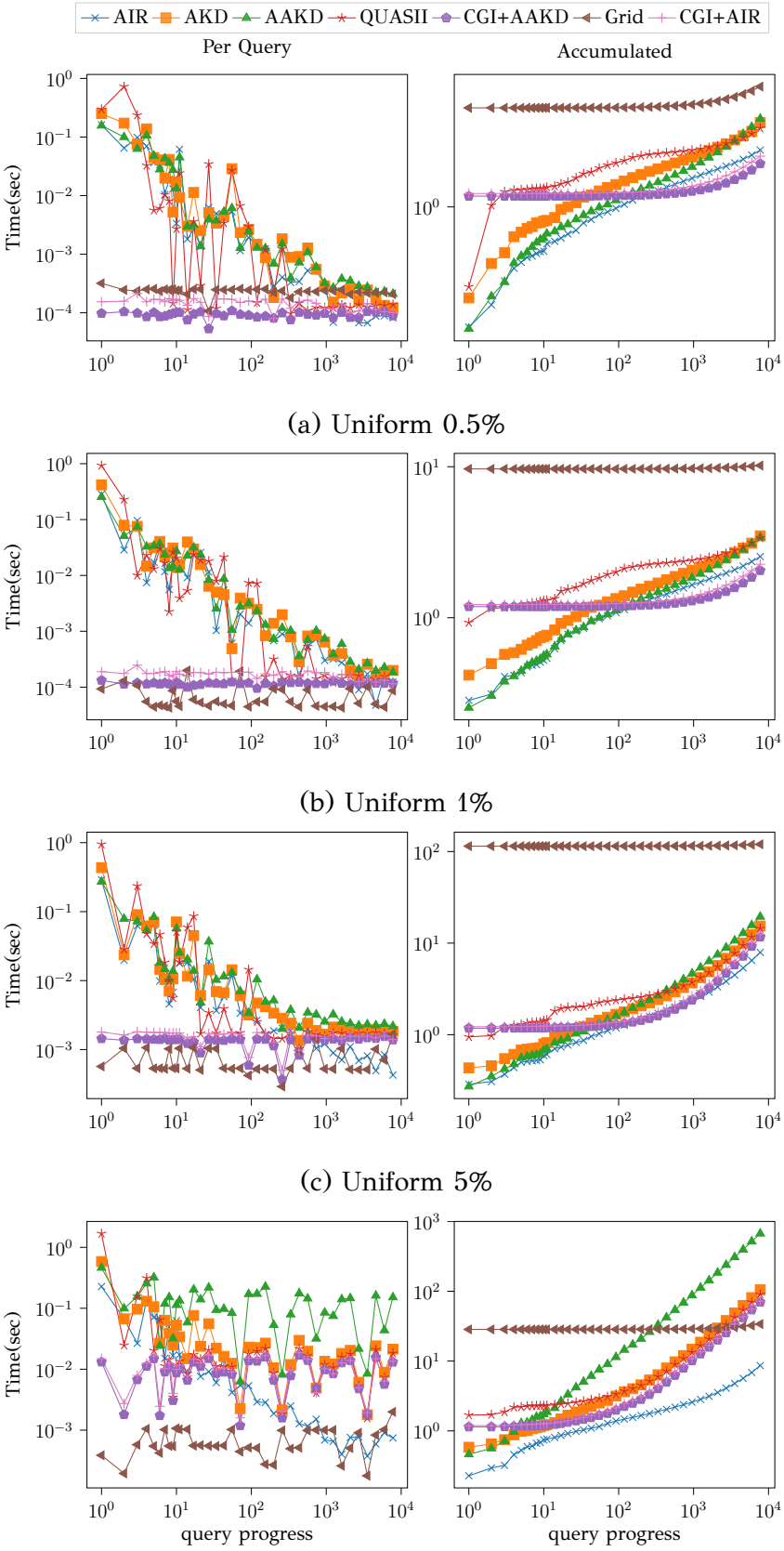
technique. Second, the heuristic employed by AAKD aims to create partitions that are square-like by cracking on the longest axis on mediocre bounds. This combination limits the pruning ability, resulting in more data being scanned per query and, consequently slower search times. If the data were handled as actual shapes then we would expect a performance more similar to AIR.

4.3.3 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. When an object with a large extent—such as one covering a substantial portion of the domain (e.g., half of the domain)—is present, the query window is excessively enlarged. This can lead to an increased number of false positives, thereby degrading the efficiency of the algorithm. To investigate the effect of this parameter, we take a closer look at the datasets we have been using.

Table 5.1 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 indices 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 4.6. 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 4.2.1. Figure 4.9 shows the performance of the different indices under these settings.

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



(a) Uniform 0.5%

(b) Uniform 1%

(c) Uniform 5%

(d) Exponential

Figure 4.9: Effect of object extent (2D shapes)

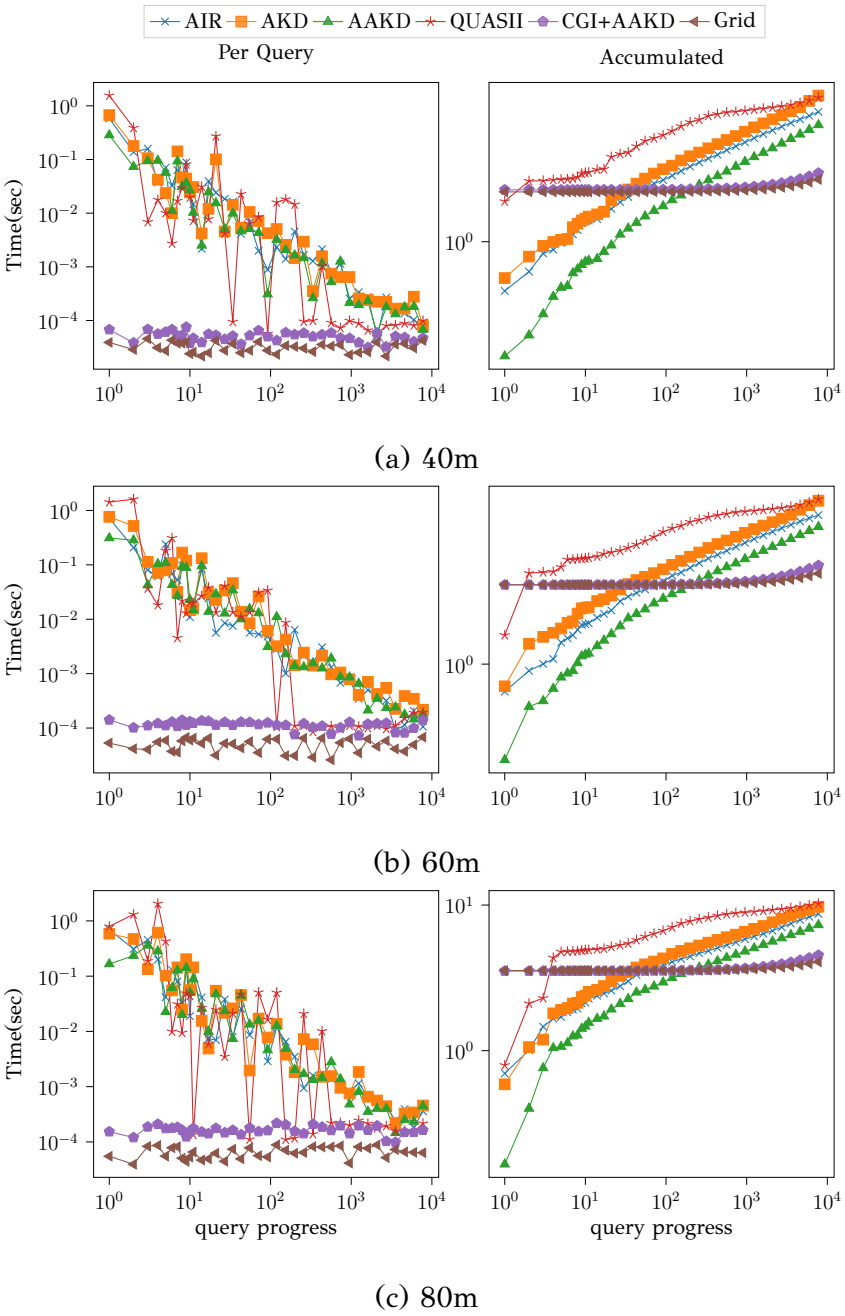
performance of AIR remains immune to changes in the size of the data objects. AKD, AAKD, and QUASII get more and more hindered with larger data objects as a consequence of the extensions. An interesting pattern to notice is the behaviour of AAKD when dealing with particularly large objects, as in Figure 4.9d, and even in the ROADS dataset in Figure 4.8d. 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 indices 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.

4.3.4 Effect of object cardinality

Figure 4.10 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 [21, 18, 22], the results demonstrate that data 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.

4.3.5 Effect of query selectivity

This subsection includes experiments showcasing the effect of query selectivity on point data. Figure 4.11 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



(a) 40m

(b) 60m

(c) 80m

Figure 4.10: Effect of data cardinality on point data

size. Overall, the **coarse granular index**, or equivalently the **static grid** performs reliably even while affected by the query size.

4.3.6 Effect of query pattern

Figures 4.12 and 4.13 show the performance of the methods in workloads with non-random access patterns for points and shapes respectively. Please note that the number of queries are lower for these experiments. 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.

The decreasing trend seen in the per query times are expected for adaptive indices. For the static indices there is also a decreasing trend in the zoom in patterns. This is due to the drop in the extent and selectivity of queries as we zoom in.

This pattern also makes the benefit of accessing the data only once become particularly evident. In the point experiment, we can also notice that since we are only accessing a few cells of the grid, and less and less 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 4.3.3, 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.

4.3.7 Effect of dimensionality

Figure 4.14 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.

4.3.8 Memory usage

Table 4.7 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.

Table 4.7: 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

4.4 Conclusions & Findings

We evaluated the performance and robustness of multidimensional adaptive indices in a plethora of diverse scenarios. Their effectiveness can be influenced by various factors such as object size and distribution, workload pattern, and more. Our comparison includes an advanced AKD implementation which adopts cracking heuristics from AIR [21]. We also included a first-time implementation of multidimensional coarse granular indexing (CGI) [25].

Grid-based methods perform best on uniformly distributed data, but they are also robust for non-uniform point data, however, their excellent performance is limited to low dimensional spaces. Highly clustered collections of real shape data collections, such as ROADS, pose challenges due to their unique characteristics, such as relatively high object sizes and high density of data in small areas. Query-window extension negatively impacts AKD-based methods such as AKD, whereas methods designed to ingest data with extent such as AIR are more resilient. To that end a hybrid approach combining CGI with AIR was tested but did not show any significant improvements.

For shape data, AIR is unaffected by object size, while CGI+AAKD maintains good performance up to 1% object size. Object cardinality has minimal impact on index efficiency; across datasets containing 20M to 80M objects, all indices maintain stable performance. Static grids and coarse granular indices are particularly robust for point datasets, showing little variation as dataset size increases.

Query selectivity significantly affects adaptive indexing. When selectivity is high, adaptive indices struggle to filter future searches effectively, leading to degraded performance. Low selectivity also impacts performance but to a lesser extent. In contrast, static grid methods maintain stable and reliable performance regardless of query selectivity. Similarly, query patterns play a crucial role in index performance. Adaptive indices improve as they learn from past queries, while static grids 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 for indexing methods, particularly for grid-based approaches, which do not scale well beyond three dimensions. AAKD struggles with higher-dimensional data, whereas AKD remains stable across different dimensions. AIR and AKD continue to deliver reliable performance as dimensionality increases, while AV-tree consistently underperforms due to its reliance on L_{max} distances, which does not adapt well to high-dimensional spaces.

Findings. Our main experimental findings can be summarized as follows:

- When dealing with *point* data irrespective of scale, distribution, and query selectivity, CGI+AAKD performs best.
- For shape datasets with small objects, CGI is the most effective. However, when objects are larger and less regular, AIR outperforms the other methods.
- In higher-dimensional spaces (more than 3D), AKD and AIR are the best choices, delivering comparable performance.
- For irregular query patterns, there is no single best method. Among the four indices discussed, none consistently achieves optimal performance across all patterns. However, AIR and AAKD demonstrate the most robust performance overall.

Our key conclusion is that a static index, the grid, which had not been considered as a competitor of adaptive indices in any previous study, is highly effective on point

data, offering robust performance across a variety of settings. Our findings challenge the assumption that adaptive indices always outperform static ones for a reasonable workload. One of our proposed extensions is also based on this method: the coarse-granular index. The intention was to create a *coarse* grid that will be refined via adaptation during the query workload. However, during tuning, the best-performing grid sizes were equal. Hence, we found that the simplicity and inexpensive build of the grid is hard for adaptive indices to overcome. Nonetheless, on shape data, which incur replication in grids, the inherently shape-oriented indices like AIR perform the best, and especially so on oddly-sized shape data.

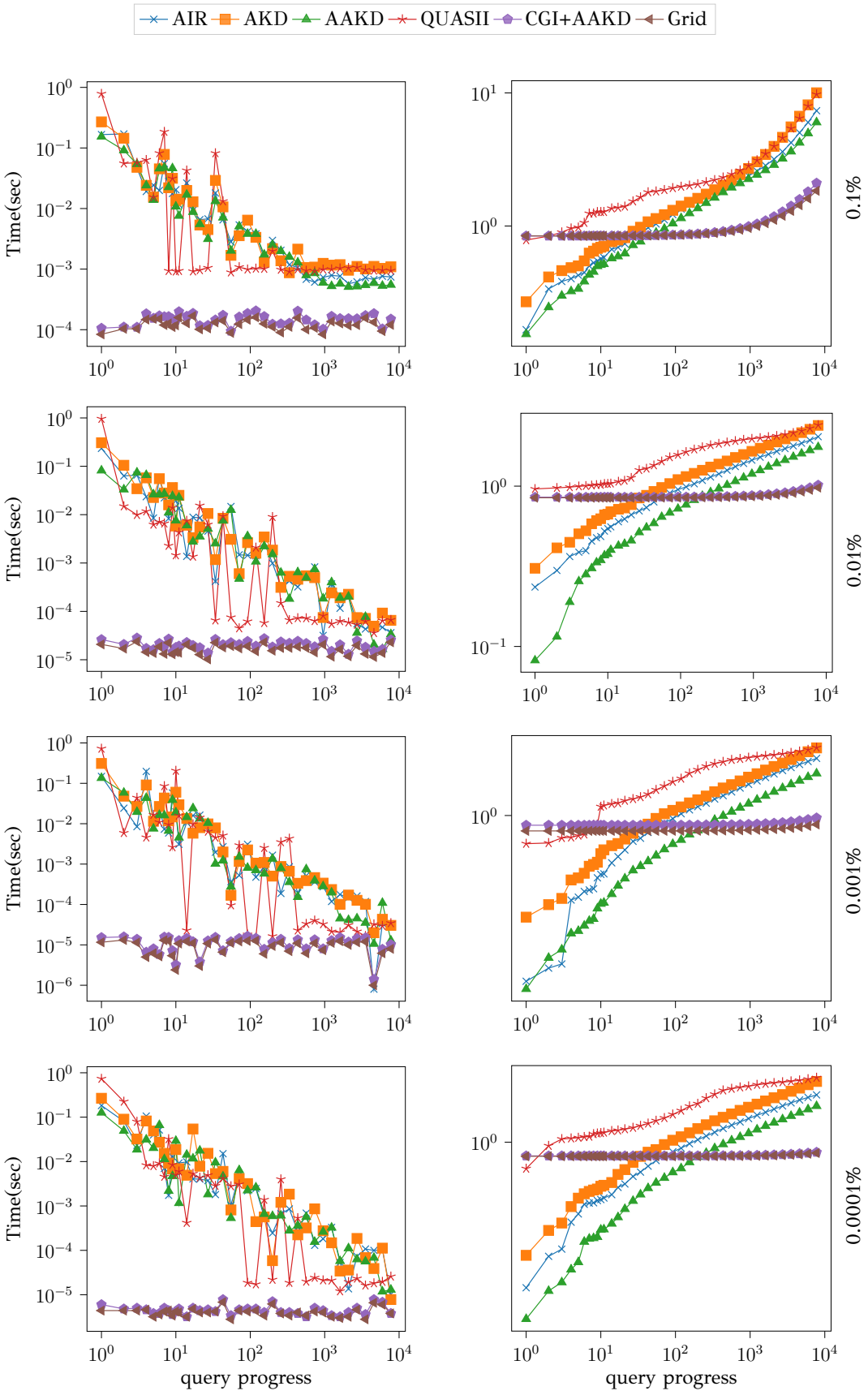
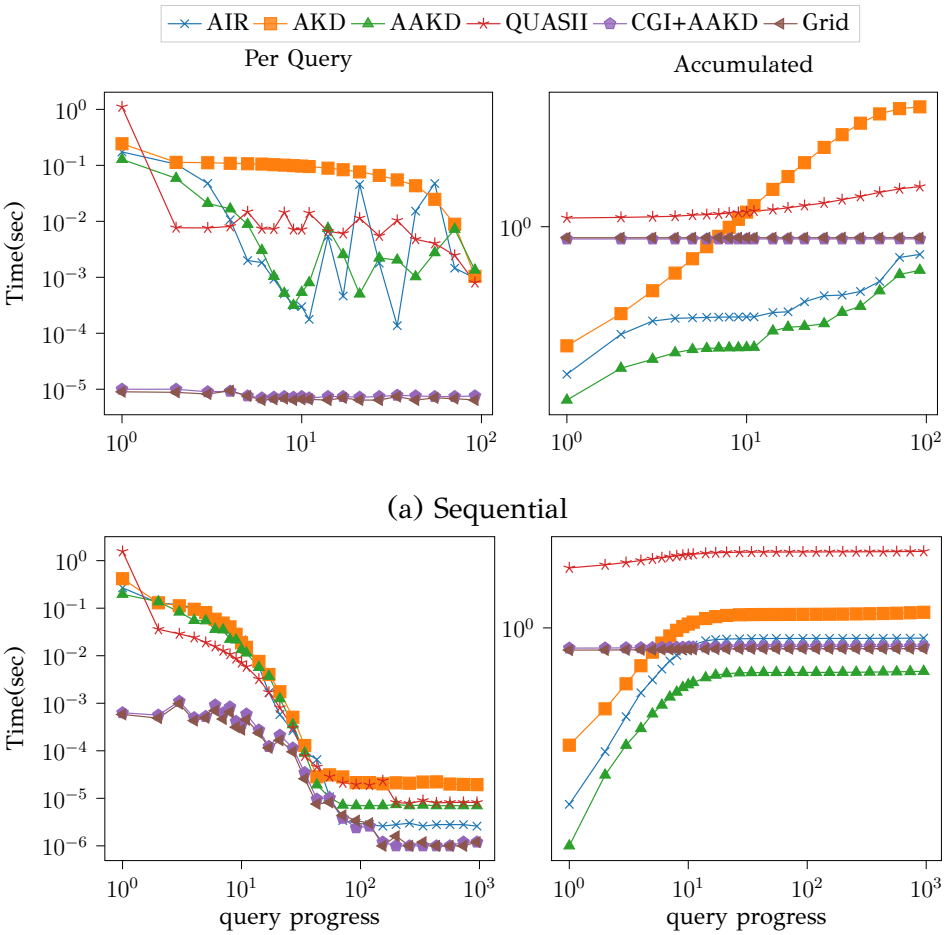


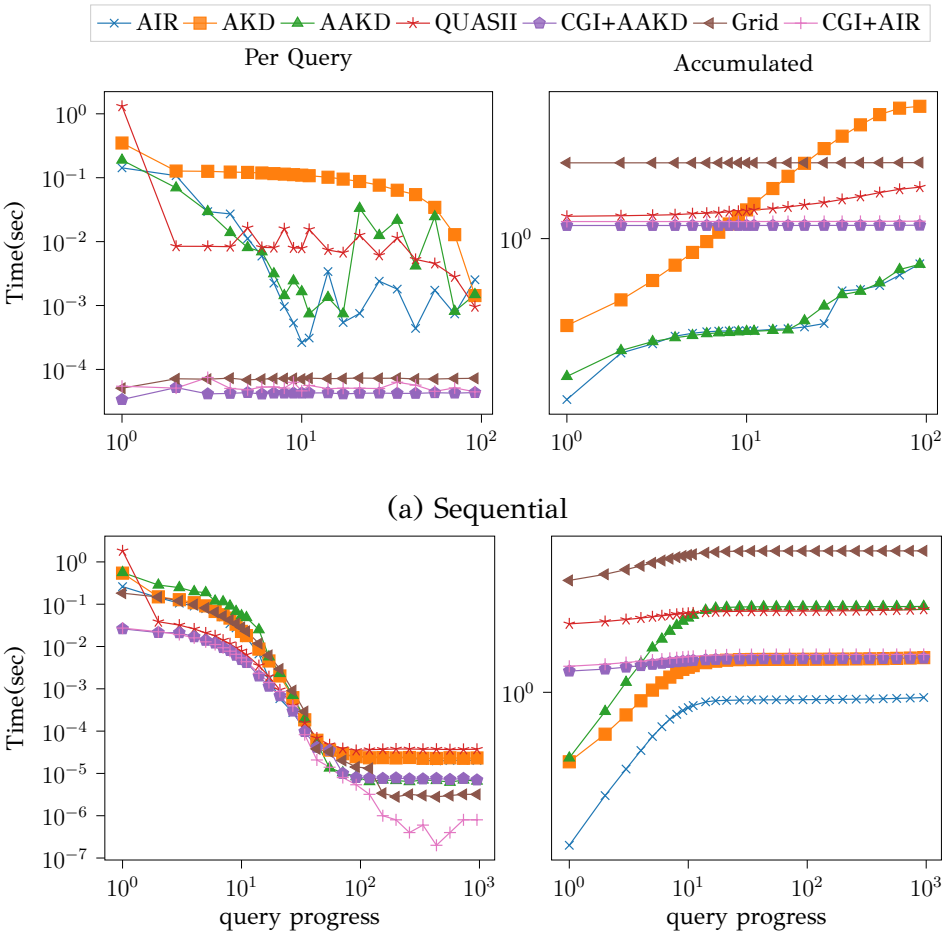
Figure 4.11: Effect of query selectivity, per query (left) and cumulative time (right).



(a) Sequential

(b) Zoom In

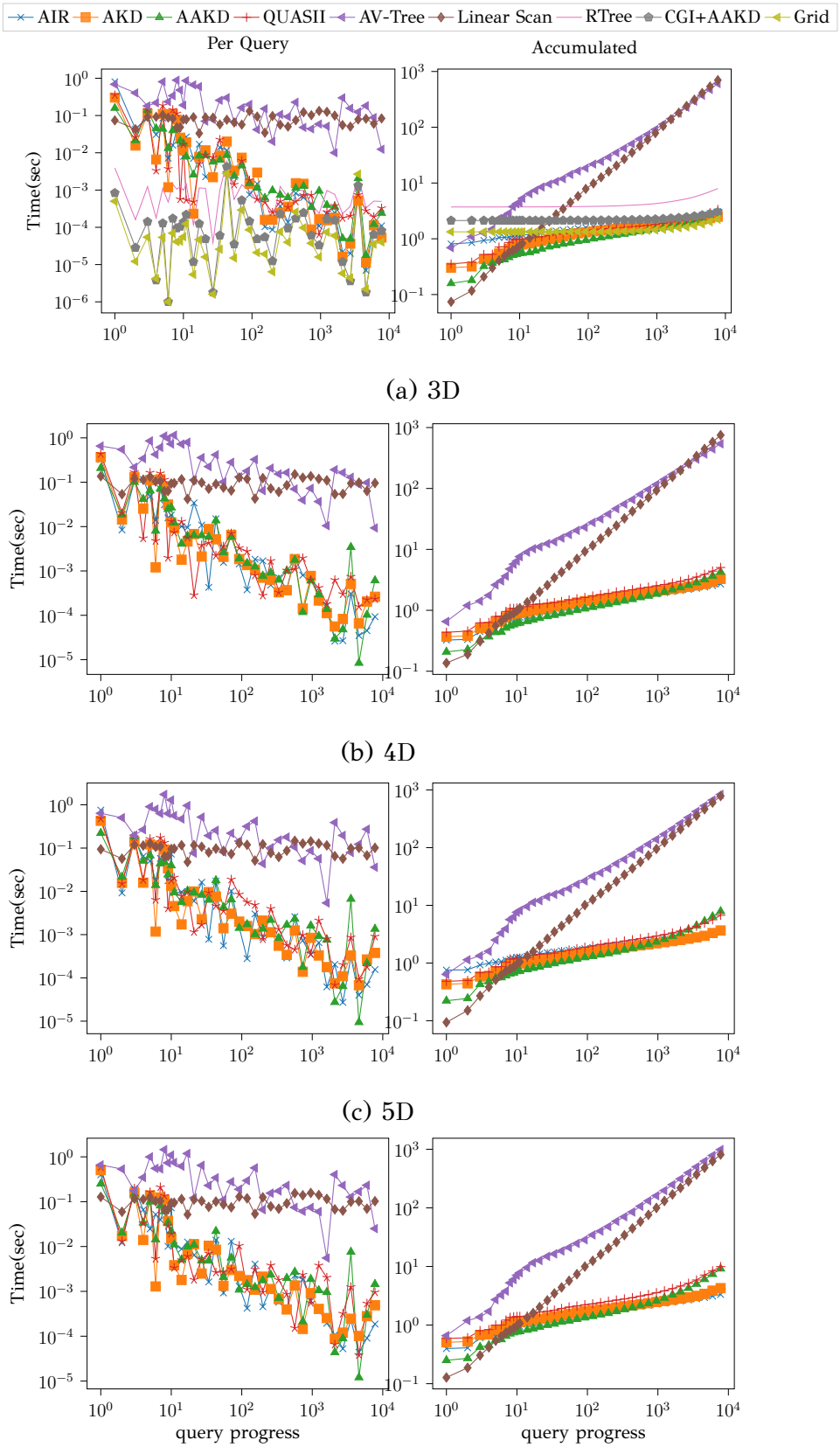
Figure 4.12: Effect of query access pattern, uniform point data.



(a) Sequential

(b) Zoom In

Figure 4.13: Effect of query access pattern, uniform shape data.



(a) 3D

(b) 4D

(c) 5D

(d) 6D

Figure 4.14: Effect of dimensionality, point data.

Chapter 5

Updating an Adaptive Spatial Index

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

Outline. Section 5.1 explores the design space for updating an adaptive index. Section 5.2 presents the reorganization strategy for adapting the index structure to accommodate updates. Section 5.3 provides a theoretical analysis of the proposed methods. Section 5.4 presents the experimental evaluation, followed by a discussion of the findings. Finally, Section 5.5 offers the concluding remarks.

5.1 GLIDE

GLIDE is a mechanism that augments any tree-based adaptive spatial index to accommodate updates interleaved with queries. We assume that the data is stored in a single array, with the data items belonging to a leaf residing in contiguous memory space. We investigate our options with respect to *when* and *how* to ingest updates into the index and the data array. Section 5.1.1 overviews the design space for GLIDE, considering these issues. We outline strategies for handling insertions in Section 5.1.2 and discuss deletions in Section 5.1.3.

5.1.1 Design options

Figure 5.1 arranges the design options we explore along three axes and presents the arising candidates we consider.

First, we consider the design choice of what event *triggers* an insertion; we outline two options: by the *self-driven* option, we insert items to the index as they arrive; by the *query-driven* option, we keep insertions in a separate global list and materialize them only once they become relevant to a query. Second, we consider two options for how we materialize a triggered insertion; in the *complete* manner, we fully traverse the index and directly enter newly inserted items into the tree leaves; in the *gradual* manner, we accommodate inserted data in separate, pre-reserved spaces in each tree node, and distribute them in bulk among the node's children once they exceed the size threshold. Lastly, we consider three options on how to reorganize the data in the single array, which we describe in Section 5.2.

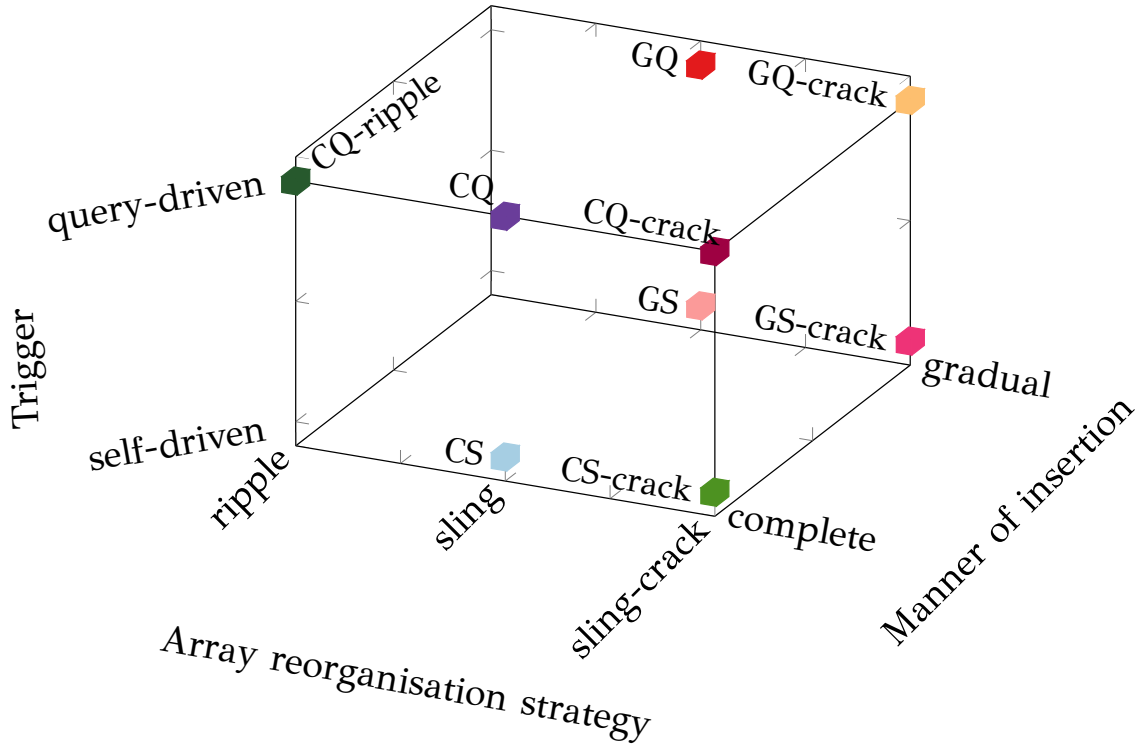


Figure 5.1: GLIDE design space.

5.1.2 Handling insertions

Complete self-driven (CS)

The *complete self-driven* (CS) insertion strategy processes each insertion fully *upon arrival*, positioning new data in the appropriate place in the tree structure, reorganizing the static array as necessary (see Sec. 5.2).

Complete query-driven (CQ)

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

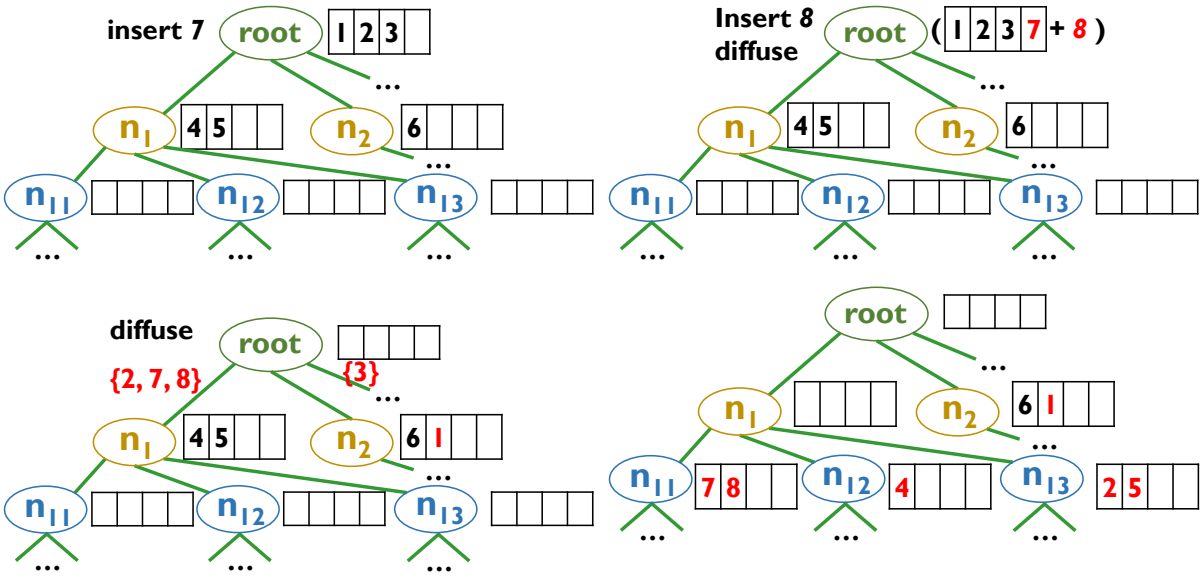


Figure 5.2: Diffusion example, tree structure.

Gradual self-driven (GS)

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

Algorithm 5.1 outlines the recursive gradual insertion method for a batch of new items to be inserted tbi in a $node$. If there is not enough space in $node$ for its spares plus tbi, diffusion takes place for all these items (Line 5). We create a tbi list of items for each child node and recursive call Algorithm 5.1 for each child (Line 12). This allows subsequently inserted items to be assigned well and query-driven cracking to operate on an up-to-date index. At a leaf that cannot fit the set of objects to be inserted among its spares, we introduce all accumulated spares and items to be inserted into the data array (Line 14), a process to be discussed in Sec. 5.2.

We show an example of diffusion in a generic spatial-index tree in Figure 5.2, assuming the tree has a θ_s threshold of 4 spare items per node. At the outset, the root holds items $\{1, 2, 3\}$, node n_1 holds $\{4, 5\}$ and node n_2 holds $\{6\}$ as spares. First, we try to insert item 7. As there is space in the root's spares, we keep it there. Then

Algorithm 5.1 Gradual insertion

```
1: procedure INSERT-GRAD(node, tbi)
2:     if node.spares.size + tbi.size < θ_s then
3:         place tbi in node.spares
4:         return
5:     if node is internal then
6:         diffused_tbi = [] for each node.children
7:         for item in node.spares ∪ tbi do
8:             b = PICKBRANCH(item, node.children) [31]
9:             diffused_tbi[b].append(item)
10:        for child in node.children do
11:            if |diffused_tbi[child]| > 0 then
12:                INSERT-GRAD(child, diffused_tbi[child])
13:    else
14:        REORGANISE(node, node.spares + tbi)
15:        return
```

we try to insert item 8. Now, there is no more space in the root's spares, hence we *diffuse* each of the root's spares and the new item down the tree, based on the tree insertion algorithm. For instance, we assign item 1 to node n_2; as there is enough space among its spares, this branch of the recursion ends here. Item 3 is assigned to one of the other children. However, items $\{2, 7, 8\}$ are assigned to node n_1, which has no spare space to accommodate them; thus, we diffuse all its spares and let each of them find its place in the next tree level.

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

Gradual query-driven (GQ)

As a fourth possible combination of our design choices, the *gradual query-driven* strategy lets insertions be triggered by queries, as CQ does (Section 5.1.2), yet follows a *gradual* manner of insertion, like GS does (Section 5.1.2). We store each arriving insertion in the unstructured log of pending insertions and, upon the arrival of a query, we scan the log to identify query-relevant objects and insert them *gradually* into the tree. We let internal nodes store a limited number of *spare* items and employ the diffusion method of Section 5.1.2; diffusion occurs as a side-effect of the query-driven tree traversal.

5.1.3 Deletions: complete self-driven

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

5.2 Reorganizing the static array

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

- the *beginning* of a leaf partition may have unused slots, or *holes*, as in [30]; we keep a counter h of such holes per leaf; hence, the contents of a leaf with range $[s, e]$ are at array positions $s + h$ to e.
- tree leaves keep linked-list pointers to their left and right adjacent leaves in the array; we note that such adjacent leaves do not necessarily have any spatial relation.

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

5.2.1 The *ripple* strategy

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

GQ. When processing a query, we denote leaves and items that overlap it as *hot* and those who do not as *cold*. We aim to keep hot items in the array but allow cold ones to leave the array to facilitate an early termination of the process. In contradistinction to a similar method proposed for indexing one-dimensional scalar attributes in column stores [30], when indexing multidimensional spatial data, array partitions corresponding to index leaf nodes do *not* need to obey a particular order. Thus, we may move leaves around liberally, taking in consideration their current position in the array and data insertion requirements.

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

1. If we insert data to the last leaf in the array, then we append the data directly to the *end of the array*.

2. If the data to be inserted fits into the *holes* at the beginning of the leaf, then we place the data there directly.

3. If k *exceeds* the size of the target leaf l, then we move the leaf along with its new contents to the array's end in $k + l < 2k$ operations; for it would take $3k$ operations to move k items from the array to the temporary log, enter k items in the array, and later return the removed k items from the log to the array.

The above cases notwithstanding, we apply the *ripple* method when inserting a number of new items of size smaller than the target leaf size. To make space for the items to be inserted, we start from that target leaf and cascade across array partitions as in [30], occupying any available holes and shifting items from the start of a partition to its end, taking over space from the next partition. Once we arrive at a *cold* leaf node, we halt the process and push a sufficient amount of data to the log. The rationale for this measure is that we are interested to keep *hot* data, relevant to the current query, in the array, but may shift cold data out of it. Otherwise, if we arrive at the end of the array and remain in the *hot* area, we expand the last leaf as necessary.

Fig. 5.3 illustrates the ripple strategy, assuming that the numbers are identifiers of spatial objects. Holes are denoted by an X. Suppose that we insert items $\{15, 16, 17\}$ into leaf i. Consider that leaves i and ii overlap the current query (i.e., they are *hot*), while leaf iii does not (i.e., it is *cold*). Leaf i has one hole, wherein we place object 15.

Then, we use the hole of the next leaf ii to accommodate $\{16\}$. Now only item 17 is to be placed. As leaf ii is *hot*, we *ripple* its contents, i.e., item 11 forward to the end of leaf ii and beginning of leaf iii and adjust the range boundaries of leaves i and ii accordingly. As leaf iii is *cold*, i.e., does not overlap the query, we cease rippling, eject cold item 13 from leaf iii into the pending insertions log (to stay there until some other query drives it back into the tree structure). Notably, by this *rippling* method, the size of the pending insertion log fluctuates.

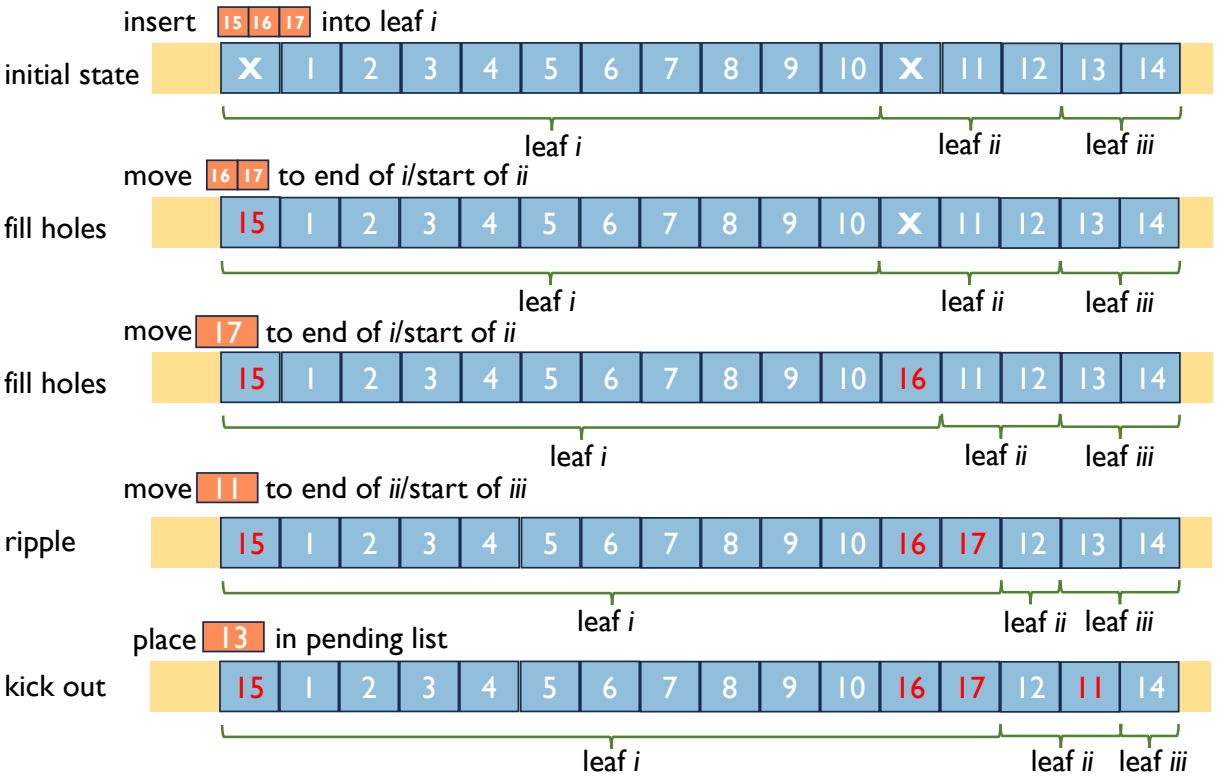


Figure 5.3: Ripple reorganisation strategy.

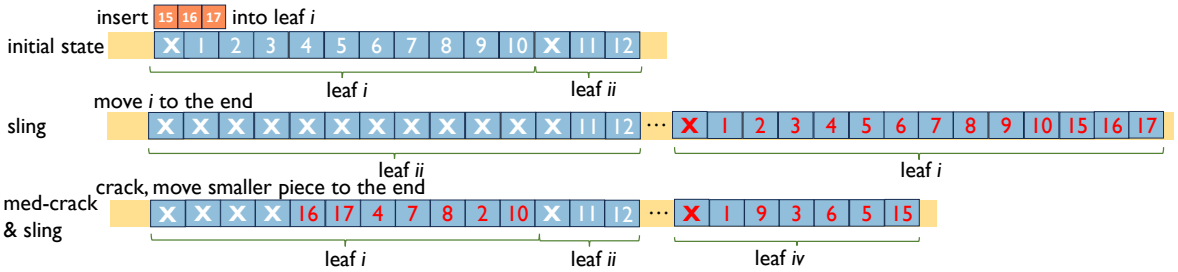


Figure 5.4: Sling reorganisation strategy: plain, with mediocre crack, with quantile crack.

5.2.2 The *sling* strategy

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

Algorithm 5.2 outlines our *sling* strategy. First, in case the target leaf is the last leaf in the array, we append the inserted data to its end (Line 2). Likewise, if there is enough available space in a leaf's *holes*, we avail of them for the insertion (Line 4). Otherwise, the sling strategy *ejects* the entire leaf to the end of the array alongside the new data and offers the empty space created by the move as holes to the next leaf (Line 8). For the sake of robustness, we endow the moved leaf with a certain amount of *default holes* Δ_H (Line 9).

Algorithm 5.2 Sling

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

Figure 5.4 illustrates the sling method with an example. We insert items $\{15, 16, 17\}$ into leaf i, which is not the last leaf and does not have enough holes to accommodate the new items. We then move leaf i along with the items to be inserted tbi to the space available at the end of the array, along with one default hole, and assign the available space left behind as holes to leaf ii.

5.2.3 Sling with a crack

By the sling strategy, an insertion to a large leaf may cause superfluous movement of data in the array and leave behind a lot of empty space. To ameliorate this effect, we amend the sling strategy with a *stochastic cracking* [17] step, replacing Lines 8–10

in Algorithm 5.2 by Algorithm 5.3. When inserting to a leaf larger than twice the regular leaf size threshold, $2M_\ell$, we crack it on a *mediocre*, i.e., the median of a small number of samples [29], to split it into two approximately equal pieces (Line 4); if at least one of those pieces leaves behind enough space for the data to be inserted, we move the smallest such piece to the array's end; if none of them leaves behind enough space, we move both to the array's end, keeping in check the amount of moved data and the memory space occupied by the data including holes.

Algorithm 5.3 Crack Upon insertion

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

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

The third row in Figure 5.4 shows an instance of the sling method with cracking in action. Considering leaf i as larger than $2M_\ell$, we choose a spatial axis and pivot as the median of 3 samples taken from the leaf, and crack items in the leaf's partition thereby. Assume items $\{1, 9, 3, 6, 5\}$ fall on the one (left) side of the pivot and items $\{4, 7, 8, 2, 10\}$ on the other (right) side by our spatial cracking criterion. As the two pieces have equal size and the tbi items fit in it, we move the first of the two pieces to the array's end. We then crack tbi items on the same pivot. Let item $\{15\}$ fall on the left side and the rest on the right side. We leverage the holes created by slinging $\{1, 9, 3, 6, 5\}$ to the array's end to place tbi alongside each cracked partition, creating the new leaf iv, which we add to the tree structure with one initial hole (Δ_H = 1). We experimented with more sophisticated choices for the cracking pivot, but did not find a better option.

We combine the designs of Section 5.1 with the reorganization strategies discussed here to create indexing methods. Standard methods use the *sling* strategy (Section 5.2.2) without extra cracking, denoted with the suffix *-sling*. We refer to combinations with the *ripple* reorganization strategy (Section 5.2.1) by the suffix *-ripple and to those with the *sling* strategy (Section 5.2.2) with a mediocre crack by the suffix *-crack.

5.3 Theoretical Analysis

We analyse the cost of insertions and queries separately, and combine results. An analysis of adaptive indexing in 1D is available in [29]; its results are applicable, mutatis mutandis, to the multidimensional case. In adaptive tree indices, the query response comprises *index traversal* and *index extension*. In the 1D case, traversal is done on a binary search tree with logarithmic complexity. In balanced trees, queries need $O(\log N + T)$ operations for N data objects and T query results. As GLIDE is implementable on top of any adaptive tree index, let the expected tree traversal operations be $\Psi(N, d, T)$ for queries on N d-dimensional data objects, each yielding up to T results. Also, let the expected total number of operations for index adaption be $\Gamma(N, r, d)$. Then the expected number of operations is $\lambda(N, r, d, T) = \Gamma(N, r, d) + r\Psi(N, d, T)$.

The complete self-driven (CS) design follows the same querying strategy, and will

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

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

5.4 Experimental Analysis

5.4.1 Implementation

AIR. We implemented[1] all different design options on top of a query-adaptive R-tree, AIR [21]. We tailored the GLIDE module to this particular structure as follows.

First, by all candidate designs (§5.1.1), new data do not enter not only the data array, but also the tree structure. Regarding leaf-splitting, if the leaf that receives the inserted data is *irregular*, then we simply allow the index to accommodate the new data. If the leaf is *regular*, i.e., has size below the cracking threshold, and in effect exceeds that threshold, then we switch the leaf to *irregular*, allowing it to be *cracked* again by future query-driven cracking operations. Thereby, the index structure accepts updates with little overhead. As we trickle down the tree on the path to the location where newly inserted data is to be introduced, we extend the MBB of each encountered node to accommodate that data.

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

Second, when cracking a leaf node by AIR [21] procedures, we disperse its spares among the ensuing pieces by the R-tree insertion heuristic [31], always choosing the cracked piece that undergoes the least area enlargement.

Third, regarding deletions, we deliberately do not tighten MBBs to reflect deletions, to make the procedure more lightweight, in the same spirit as lazy-update R-trees [98]. As deletions are relatively rare compared to queries and insertions, we anticipate that eschewing the tightening of MBBs benefits overall GLIDE performance; node MBBs subjected to deletions are eventually tightened due to queries and insertions.

AV-tree. To evaluate the generality of the proposed methods, we apply them on the Adaptive Vantage Tree (AV-tree) [22], an adaptive index structure designed for indexing high-dimensional data in metric spaces. The AV-tree partitions the space around query centers into units defined by hyperspheres, utilizing distance bounds. It supports both range and k-nearest neighbor (kNN) queries. Since we evaluate range queries on AIR, we focus on kNN queries on the AV-tree to cover more of the spectrum of query types.

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

5.4.2 Experimental setup

We conduct an extensive experimental study to evaluate GLIDE using real and synthetic data on realistic workloads. We implemented GLIDE and competitors in C++ and compiled them in g++ 7.4.0 with the -O3 switch; experiments ran on a 10-core Intel Xeon machine at 3.10GHz with 396G RAM running Ubuntu 18.04.3 LTS.

Performance measures. Following the common practice in prior work [20, 19, 2, 1, 30], we measure the progressively evolving response time during the workload. In terms of response times, we measure: (i) the cost per query over a workload, averaged over 5 runs; and (ii) the cumulative cost, which aggregates the cost per query over a workload; including the creation time for static indices. For the sake of

fairness, all methods perform identical count range queries. Insertion times inevitably fluctuate, as some trigger cascading diffusion; to visualize results comprehensibly, we add a continuous moving-average line in those plots, with window size 30. In cases where the full workload progression offers no new insights, we show only the time to evaluate the entire workload.

Table 5.1: Data sets.

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

Datasets. In experiments where GLIDE is applied on AIR, our focus is on the adaptive indexing of spatial objects. We generated a large 2D synthetic dataset of 64M rectangular shapes using SpiderWeb [99]. The location, as well as the width and height of these objects adhere to a uniform distribution within the [0, 1] and [0,0.01] range, respectively. We also experiment with publicly available 2D and 3D real datasets:[2] ROADS and EDGES from the US Census Bureau and BUILDINGS from OpenStreetMap. The ROADS data feature shapes of U.S. roads and the EDGES data comprise of lines on the U.S. map, including roads, rivers, and borders. The BUILDINGS dataset is comprised of the boundaries of all buildings worldwide. We also use a real-world 3D data set of taxi cab trip records[3] from year 2010 normalising pick-up and drop-off longitudes, latitudes, and timestamps to represent 3D boxes. Finally, we use the popular MNIST[87][101], database of handwritten digits, to test the application of GLIDE on AV-tree. We used UMAP [91] to reduce their dimensionality down to 50.Table 5.1 summarizes data characteristics.

Table 5.2: Query workloads.

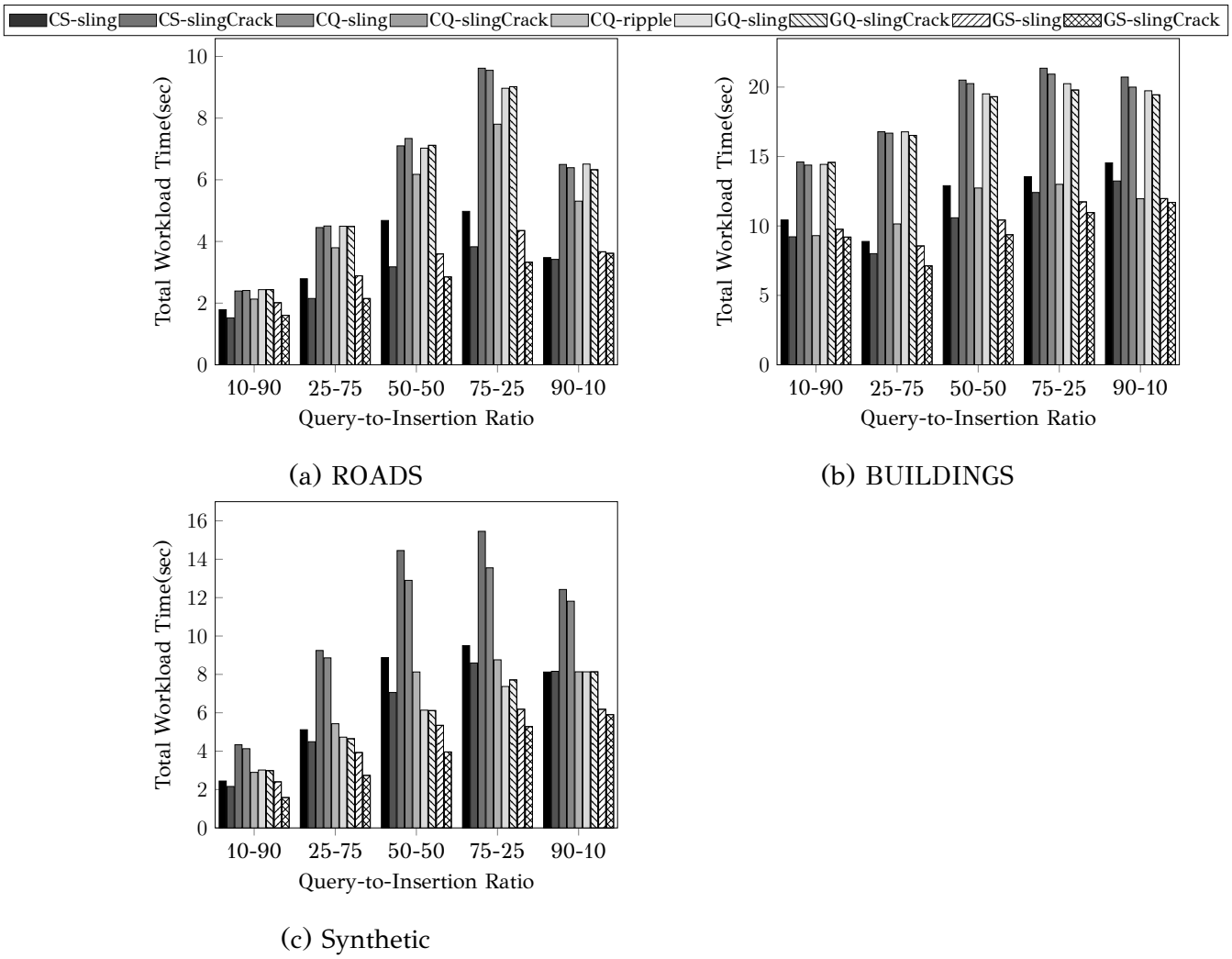
Location distribution	*Size distribution*	*Source*	*Size*
Uniform	Uniform	Synthetic	100k
Zipfian	Uniform	Synthetic	100k

5.4.3 Workloads

We measure time per query and cumulative time over workloads of queries intertwined with insertions. For the range workloads, we use **query workloads** consisting

[2] `http://spatialhadoop.cs.umn.edu/datasets.html` at the University of Minnesota [100]

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



(a) ROADS

(b) BUILDINGS

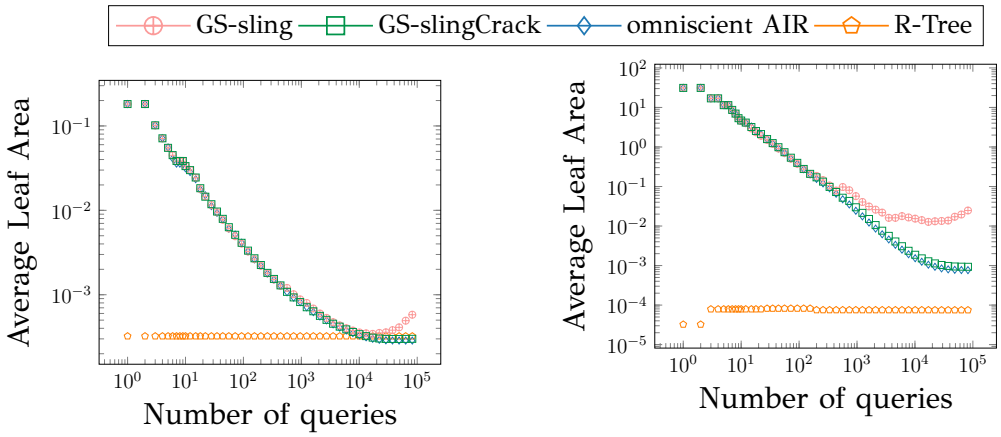
(c) Synthetic

Figure 5.5: Ablation study on range workloads.

of at most 100K rectangular queries placed according to either a Uniform distribution or a Zipfian distribution with $\alpha = 4$ using Python's scikit-learn [92] module. Table 5.2 summarises workload characteristics. As the default option, with the ROADS and EDGES datasets, we tailor each query extent so that it has a result size in the order of 0.001% (e-3%) of all data objects. With the BUILDINGS dataset, which features locations and shapes of buildings, to create queries of the desired selectivity on built areas, we select at most 100K random objects from the data and extend their width and height. While we use a default query selectivity of e-3%, we also look into the effect of the query result size by investigating other values: e-4%, e-2%, and e-1%. We interleave queries with insertions as follows.

Each workload performs 100K actions encompassing a shuffled combination of insertions and range queries, with a set ratio between the amount of queries and insertions. We draw the inspiration for this class the YCSB benchmark workload type

E [102], a well-established Yahoo! data management system benchmark. The default ratio of queries to insertions is 75% range queries to 25% insertions (i.e., 75K queries and 25K insertions). To draw data for insertion, we set aside a properly sized subset of the data set at hand *apart* from the fixed data set sizes reported in Table 5.1. Moreover, as the query-to-insertion ratio is seldom known a priori. To affirm robustness with respect to variations in that ratio, we examine a range of query-to-insertion ratios other than the default one. We dub these query-and-update workloads, ***action ratio*** workloads.



(a) Synthetic, Leaf Areas

(b) ROADS, Leaf Areas

Figure 5.6: Average Leaf Areas on range workloads

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

Table 5.3: Parameters, uniform 2D shape data, 25% inserted.

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

5.4.4 Parameter Tuning

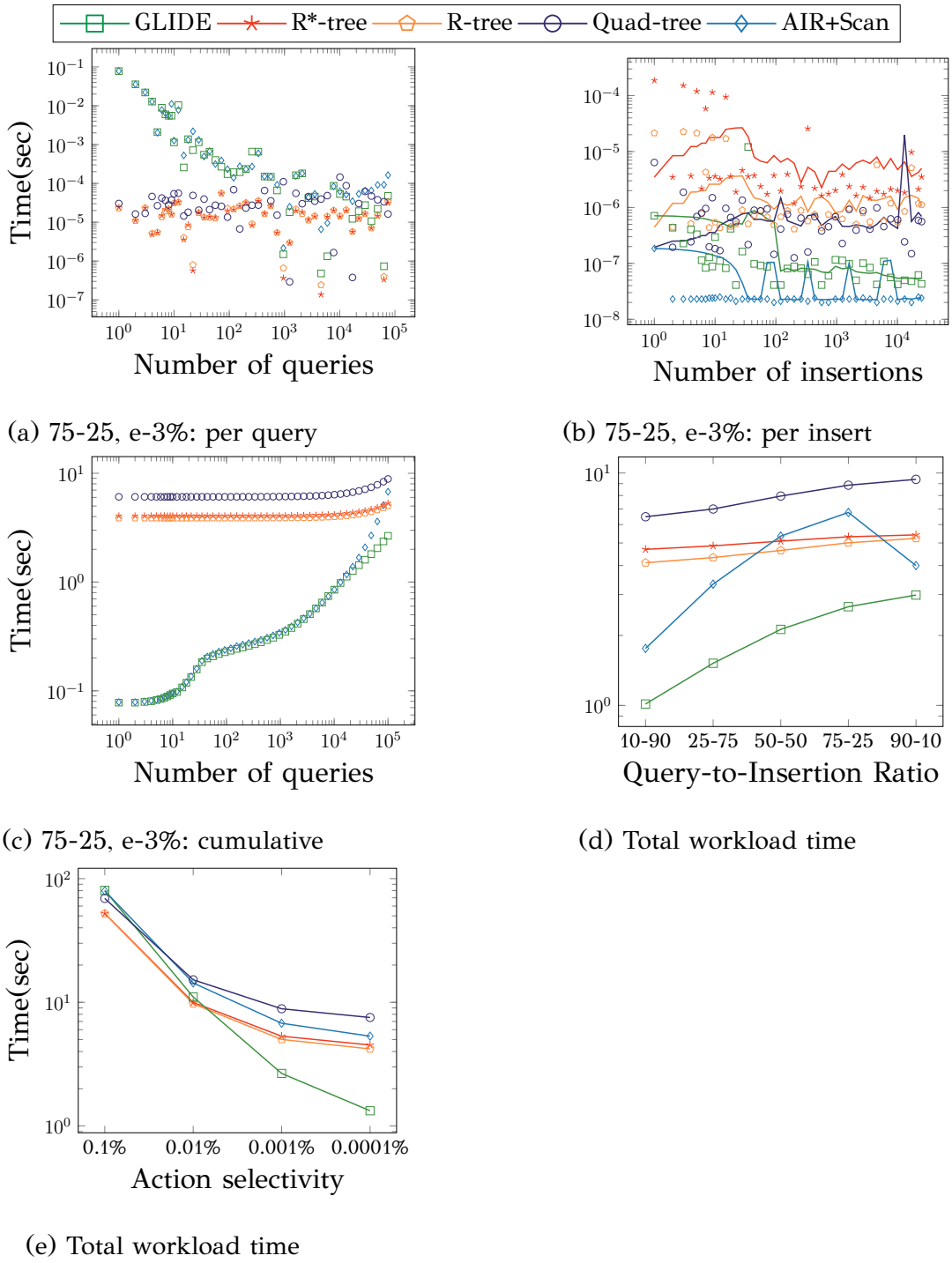
We use the range workloads on AIR to investigate the parameter values. Firstly, we investigate the choice of values for GLIDE parameters: regular leaf size threshold M_ℓ, tree fan-out f, default number of holes Δ_H, and limit of spare items stored in nodes θ_s. For the first two, we use the values found best in [21], i.e., $M_\ell = 64$ and $f = 16$. Regarding Δ_H and θ_s, Table 5.3 shows the total time (in seconds) spent by GLIDE on the whole Uniform workload of queries and insertions of various ratios with different configurations. Following these results, we choose to allow 64 holes on moved leaves ($\Delta_H = 64$) and to up to $\theta_s = 8$ spare items in each internal node, by virtue of the dependability of these values. We stress that the performance of GLIDE is not overly sensitive to parameter values. For the kNN workloads in the AV-tree we use the same Δ_H and θ_s values, and let the cracking threshold parameter, θ, to be set as 128 as suggested in [22].

5.4.5 Ablation study

We perform analysis of the designs on the range workloads implemented on the adaptive R-tree. We compare GLIDE variants differentiated by the trigger and manner of insertions and the array reorganisation strategy. We use each on a synthetic dataset of 8M objects, the ROADS data, and the BUILDINGS data, with workloads of varying action ratios represented on the x-axis. We use a shuffled mix of queries and insertions as described in § 5.4.3 with ratio of 75 to 25 respectively.

We compare the total workload runtime of the variants presented in Section 5.1, including CQ-sling, CQ-slingCrack, CQ-ripple, GQ-sling, GQ-slingCrack, GS-sling, GS-slingCrack, CS-sling, and CS-slingCrack.

Figure 5.5 presents our results. Observe that GS-slingCrack achieves the best cumulative time in insertion-heavy workloads and proves to be robust against various action ratios. Query-driven methods are all burdened with keeping an extra array for *pending* new items, which they need to scan with each query, whereby query-driven GLIDE variants fully or gradually insert the related items into the index. In such query-driven approaches, insertions are cheap whereas range queries can be expensive. One might intuitively expect such variants to perform worse on insertion-heavy workloads, as they need to scan through more inserted items; however, such workloads invoke fewer queries that warrant those expensive scans, causing perfor-



(a) 75-25, e-3%: per query

(b) 75-25, e-3%: per insert

(c) 75-25, e-3%: cumulative

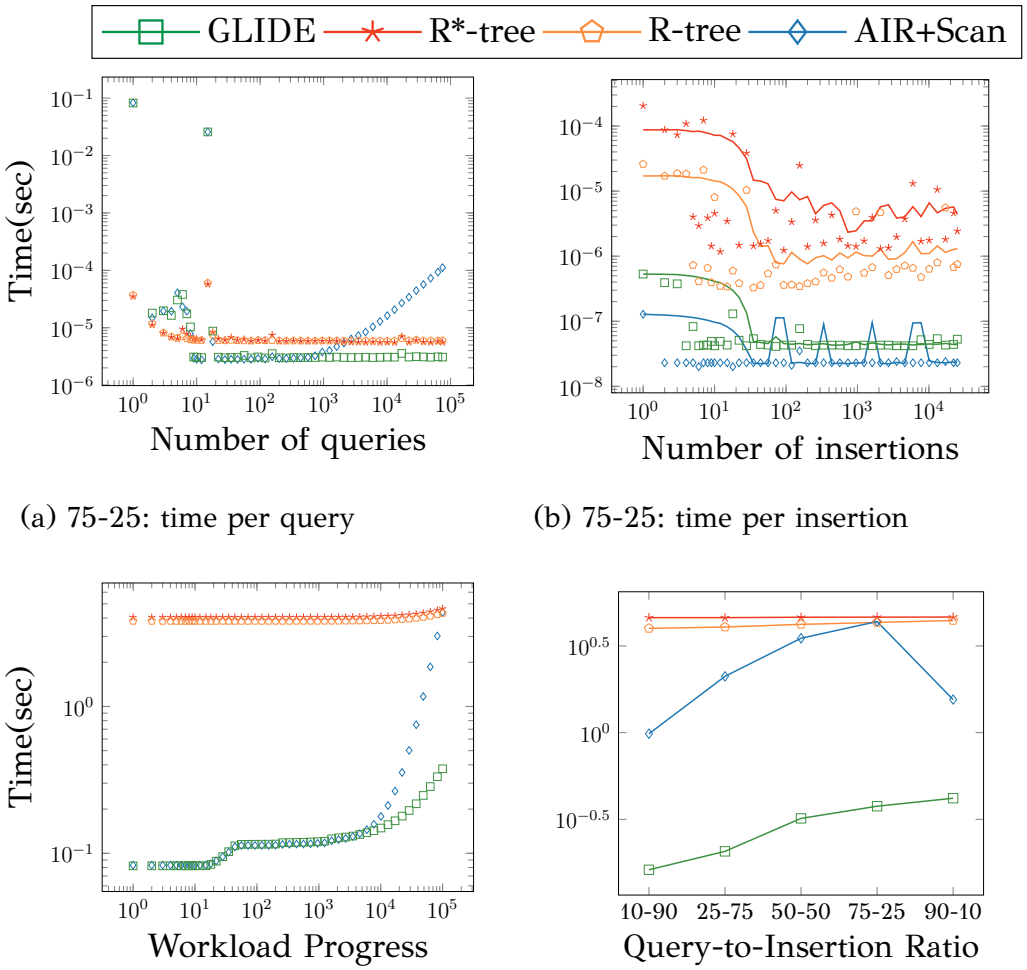
(d) Total workload time

(e) Total workload time

Figure 5.7: ROADS dataset, Uniform range queries.

mance to deteriorate as the query-to-insertion ratio tilts towards the query-heavy side; performance relapses after the unsorted list becomes small enough that its scans are less burdensome. Still, GS-slingCrack presents the most dependable performance regardless of workload arrangement.

We observe that the extra stochastic crack upon insertions drastically improves the



(a) 75-25: time per query

(b) 75-25: time per insertion

(c) 75-25: cumulative time

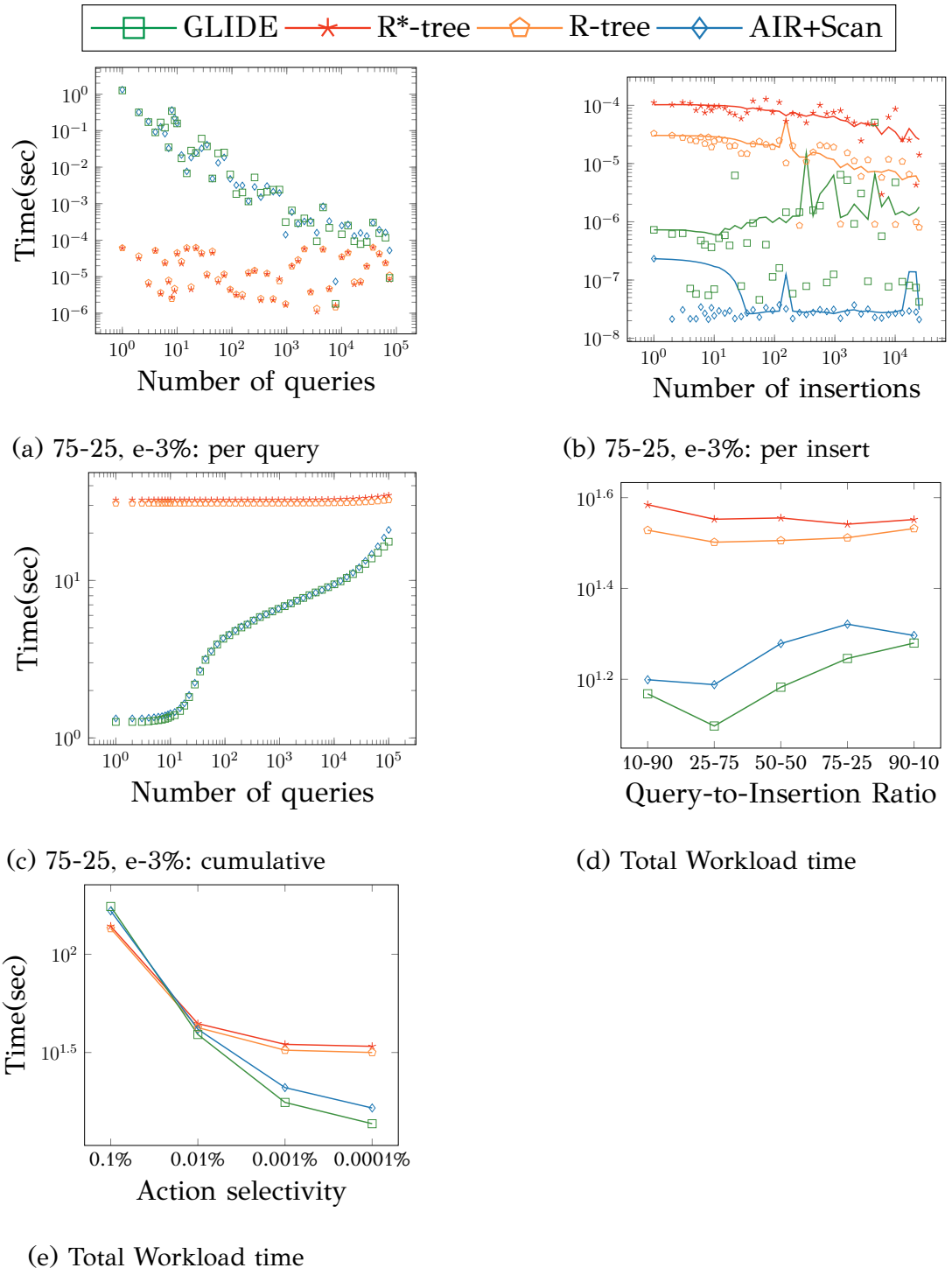
(d) Total workload time

Figure 5.8: ROADS dataset, Zipfian range queries.

performance of GS-sling. To understand this phenomenon, we perform the following experiment: We measure the average area of tree leaves as the workload progresses vs. those of an R-tree receiving insertions using the Superliminal[4] R-tree implementation, which allows setting fan-out 16 and leaf size threshold 64, as in GLIDE, and, for reference, a version of AIR with all data pre-loaded, denoted as *omniscient* AIR, which does not face data insertions. Figures 5.6a and 5.6b show our results on the synthetic shape data and ROADS data, respectively. Notably, in both datasets GS reaches large leaf areas after the 100th action, while GS-slingCrack keeps leaf sizes checked, as it mostly cracks leaves that receive insertions, and reaches average leaf size as small as the R-tree, and slightly larger than AIR, which is privileged in this comparison.

Summary. In insertion costs, the query-driven design with gradual insertion (GQ) is

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



(a) 75-25, e-3%: per query

(b) 75-25, e-3%: per insert

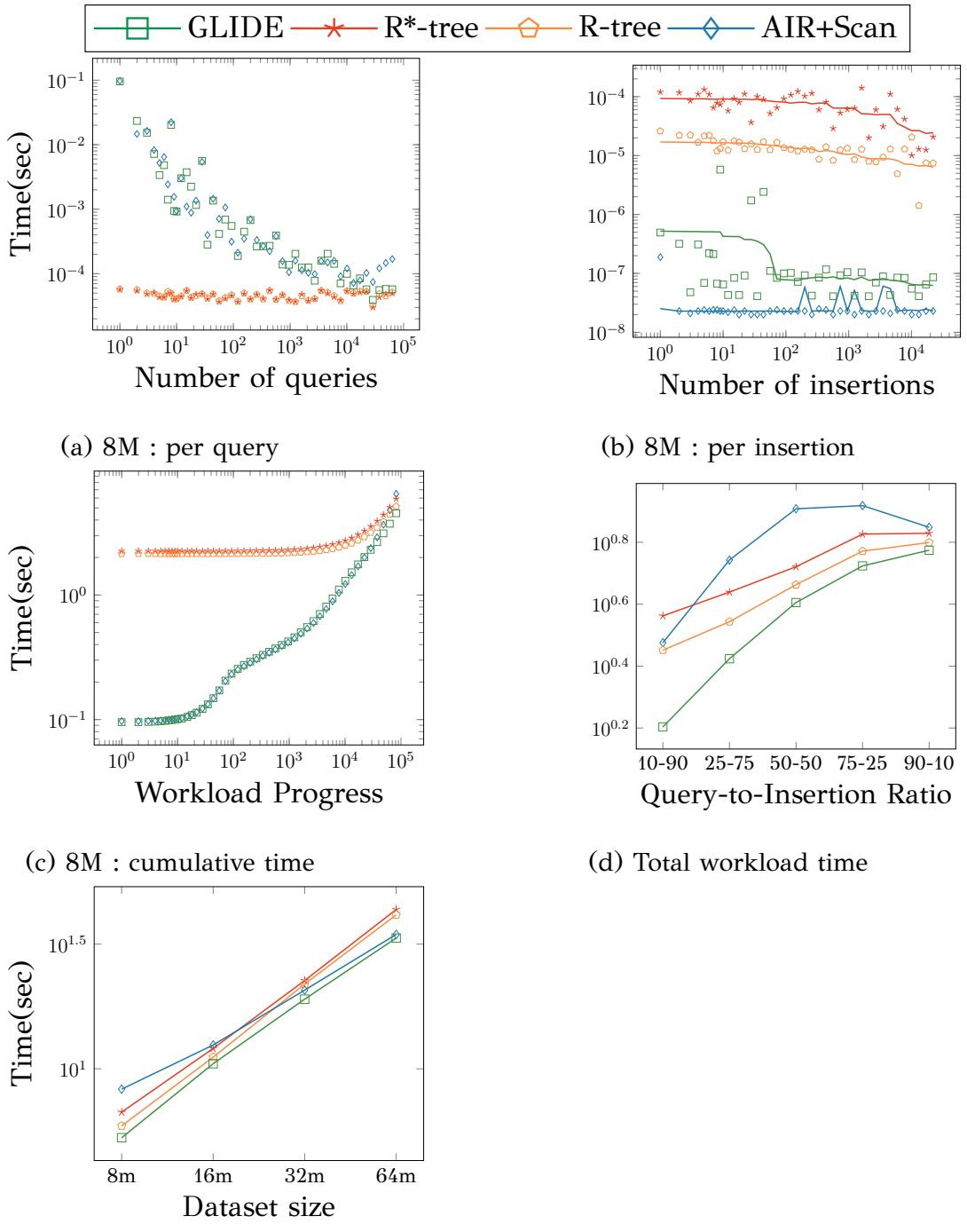
(c) 75-25, e-3%: cumulative

(d) Total Workload time

(e) Total Workload time

Figure 5.9: BUILDINGS dataset, Uniform range queries.

best. In query time, the self-driven design with complete insertion (CQ) is best. The gradual self-driven design with an extra crack upon insertion (GS-slingCrack) is the most robust option, performing best in terms of total throughput in the plethora of experimental settings we have tried.



(a) 8M : per query

(b) 8M : per insertion

(c) 8M : cumulative time

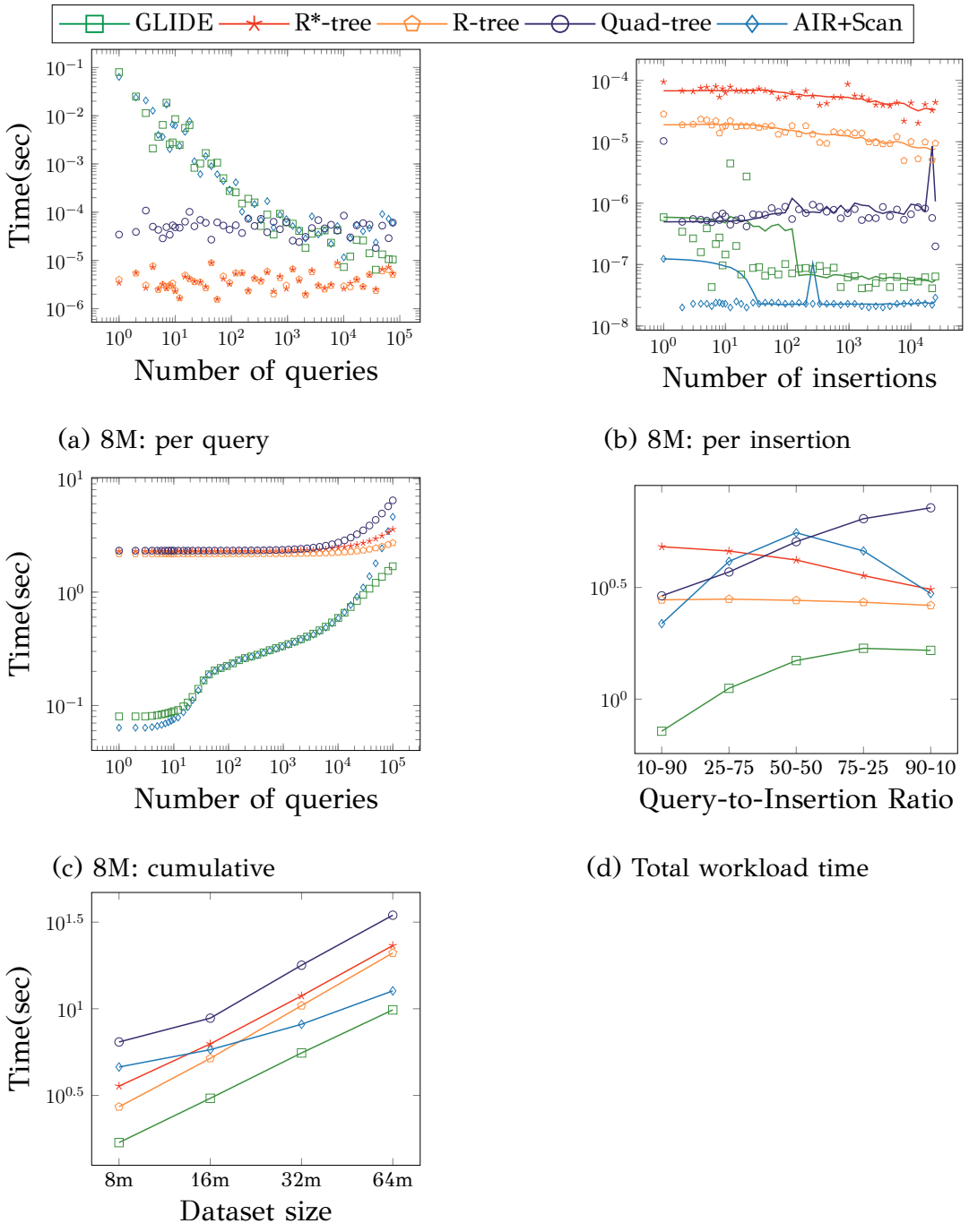
(d) Total workload time

(e) Total workload time

Figure 5.10: Uniform 2D shape data, uniform 75-25 range workload.

5.4.6 Range workloads comparative study

As our main baseline, we include a naive extension of AIR with a simplistic array, denoted as AIR+Scan, which appends inserted data to a separate log-like structure without an index. When we evaluate a query, we also scan this auxiliary data structure



(a) 8M: per query

(b) 8M: per insertion

(c) 8M: cumulative

(d) Total workload time

(e) Total workload time

Figure 5.11: Uniform 2D point data, uniform 75-25 range workload.

to retrieve results from the inserted data. Besides AIR+Scan, we compare GLIDE against implementations of the following methods:

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

- A static in-memory Quad-tree.

Both static tree implementations were taken from the Boost[5] library. We set the leaf size of R-tree variants at 16, as recommended, and conduct the comparison on different workloads of queries and updates applied to our synthetic and real datasets. The Quad-tree [34] implementation is provided in [39].

We organize the remainder of this section as follows: We evaluate GLIDE under varying action-ratios (§5.4.6), different dataset sizes (§5.4.6), and query selectivity (§5.4.6). Next, we assess performance on point data (§5.4.6) and examine how the system handles workloads that include deletions (§5.4.6). We also investigate behavior when the order of range queries is pathologically sequential (diagonal in space) (§5.4.6). Lastly, we examine performance on 3D datasets (§5.4.6).

Varying Query-to-Insertion ratio

We use shuffled action-ratio workloads that contain 100K actions, as defined in Section 5.4.3. To study the index robustness across different environments, we test a range of ratios between range queries and insertions in the workload, intertwining insertions into Uniform and Zipfian queries on the ROADS data-set. Figures 5.7 and 5.8 show our results with the x-axes of 5.7d and 5.8d representing action ratios.

To illustrate workload progression, Figures 5.7a, 5.7b and 5.7c display the trend of action times for 75-25 action ratio workloads as an example, and Figure 5.7d shows the total workload time across different ratios. The observed decreasing per query times of AIR and GLIDE are typical for adaptive indices. As Quad-tree is a space-partitioning index, it handles dynamic insertions better than the data-partitioning R-tree variants. However, that minor advantage does not counterbalance other costs in the overall workload time; this behaviour persists across different action ratios, as Figure 5.7d shows; given these results, we exclude Quad-tree from subsequent shape-data experiments.

Figures 5.8a, 5.8b, and 5.8c present the progression of the 75-25 ratio workload with Zipfian queries, while Figure 5.8d shows the trend of total workload times. In a Zipfian workload certain areas are queried more often than others, hence GLIDE builds a compact index with a few regular leaves thoroughly indexing oft-queried areas and a few irregular leaves that are rarely accessed. It achieves shorter times per

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

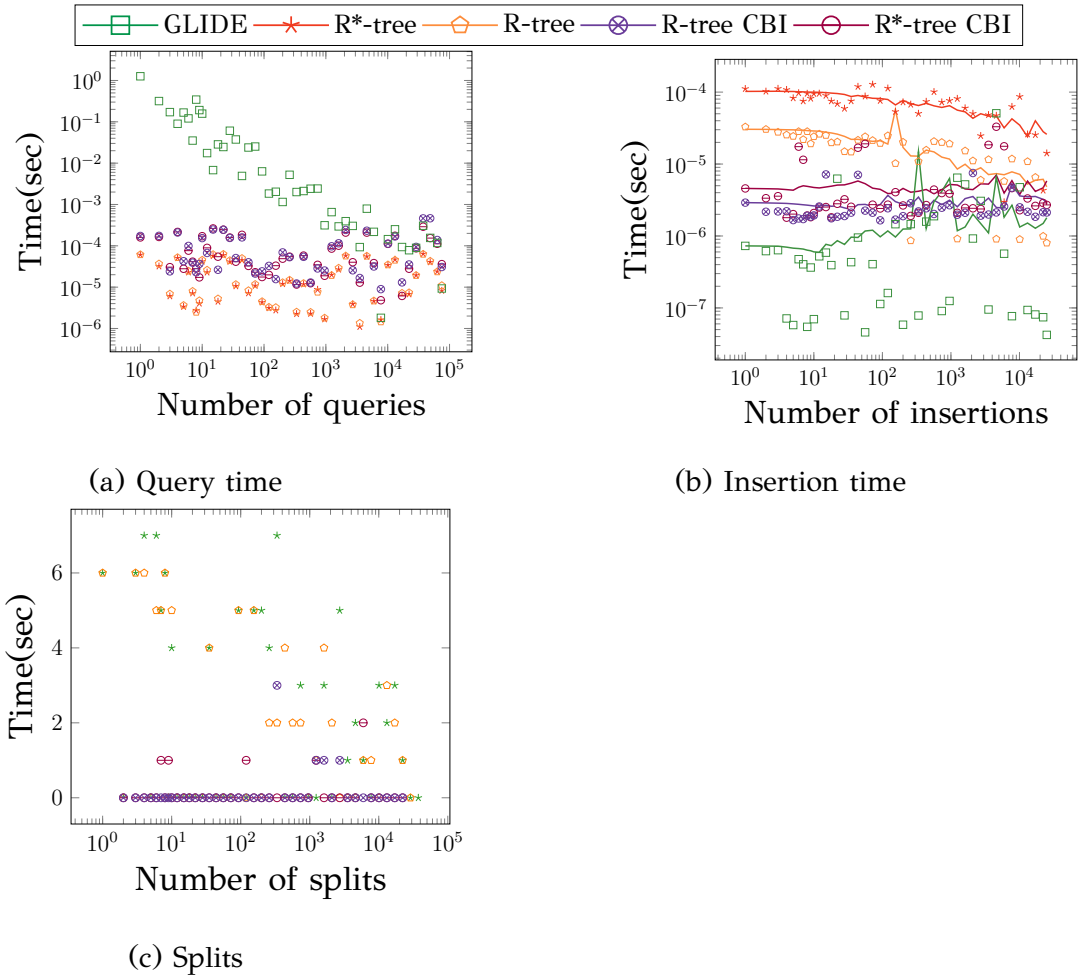
query (Figure 5.8a) and hence cumulative times. The AIR index behaves similarly, yet the extra work of scanning the newly-arrived data eventually becomes cumbersome. The insertion performance of GLIDE also outpaces classic R-tree methods (Figure 5.8b), superseding their burdensome index creation time and slower query times to result in a striking improvement in the total workload time (Figure 5.8c). This effect remains unabated by the ratio of the actions in the workload, as Figure 5.8d reveals. The trend for AIR+Scan is similar to the one observed in the ablation studies for the query-driven designs, as discussed in Section 5.4.5, and exhibits the same trends across ratios for Uniform and Zipfian queries. Figures 5.10d and 5.11d replicate this study on synthetic shape data and point data (cf. Section 5.4.6), with analogous results.

Figure 5.9 illustrates that GLIDE preserves its advantages on the BUILDINGS data, with a realistic workload and across various ratios. To inspect the separate costs more thoroughly, we present those times decoupled in Figure 5.12. Notably, as Figure 5.12b shows, insertion times of GLIDE are lower than those of the classic solutions. Moreover, the insertion times for both static R-tree variants start at high values and decrease as the workload progresses. To understand this *front-loaded* behaviour, we measured the insertion time of pre-built R-tree and R*-tree variants that are initialized on the data set by inserting data items *one-by-one*, instead of using the default Sort-Tile-Recursive (STR) bulk-loading [103] method; we denote these variants as CBI (**c**reated **b**y **i**nsertion). As Figure 5.12b shows, CBI variants exhibit stable, rather than front-loaded, insertion cost. We infer that the STR bulk-loading method builds *packed* trees that initially necessitate intensive leaf-splitting to accommodate insertions, while the space created by such initial splits suffices to absorb insertions later in the workload, depicted in Figure 5.12c. Previous work has used the term *waves of misery* [104] for this front-loaded, in general *oscillating*, behaviour of indices.

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

Varying data size

We also compared all methods, for both shape and point data (cf. Section 5.4.6), using different synthetic data set sizes: 8, 16, 32, and 64 million. Figures 5.10a, 5.10b, 5.11a,



(a) Query time

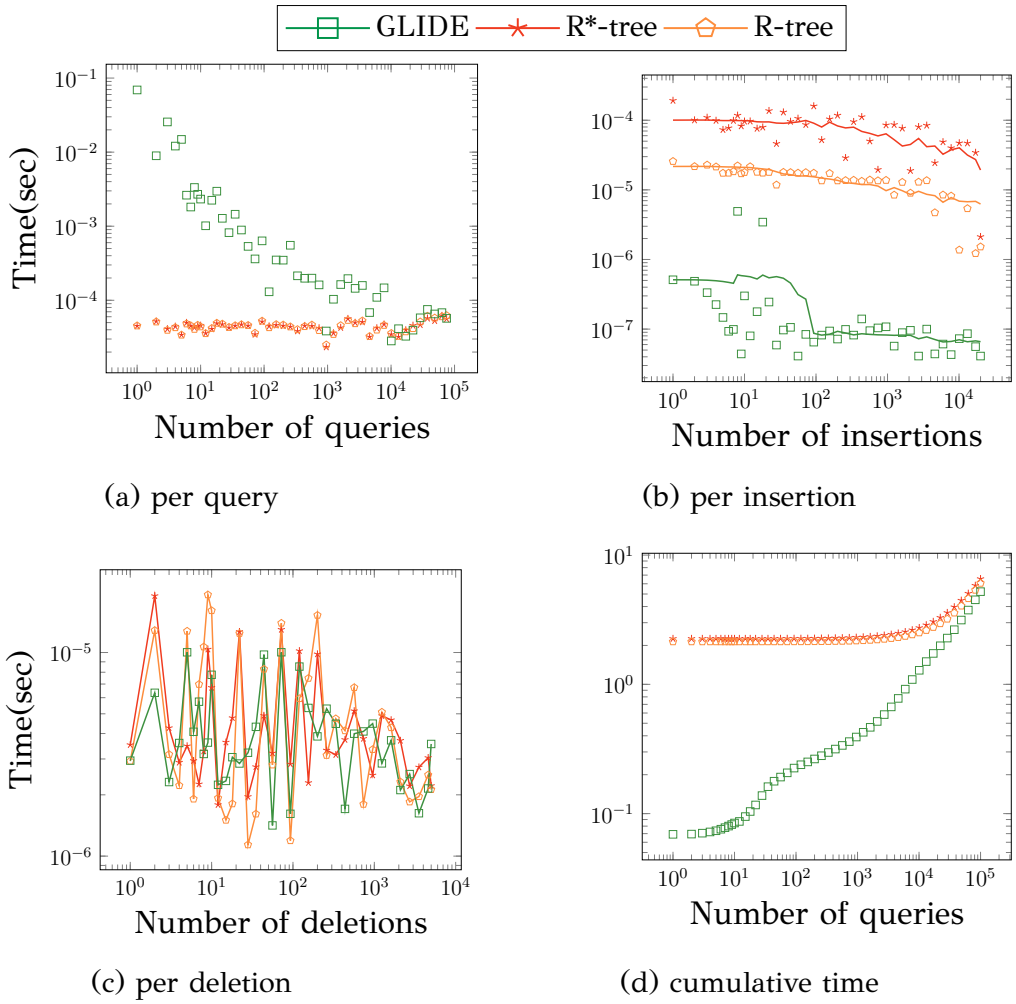
(b) Insertion time

(c) Splits

Figure 5.12: Buildings data, 75-25 per query decoupled time.

and 5.11b plot the decoupled per query and per insertion times for each case, for the 8M size experiment, while Figures 5.10c and 5.11c show the respective cumulative times to tackle the workload, which remain favorable to GLIDE throughout the 100K actions. Other dataset sizes follow similar trends. Most pertinently, as the results in Figures 5.10e and 5.11e illustrate, GLIDE scales well with dataset size, indicated on the x-axis, with both shape data and points. All methods display a linear growth of cumulative time in response to dataset size. AIR with linear scan has a less steep ascent, yet that is only due to the fact that it always scans at most 25K insertion items, while other methods manage growing indices in their insertions; this advantage is only an artifact of our specific experimental design, which keeps the number of insertions stable and only lets the initial data size grow. Naturally, it does not scale with a growing insertion workload.

Exhibiting the typical behavior of indices assembled by adaptation to queries, GLIDE starts out with high per-query cost that follows a descending trend. Eventually



(a) per query

(b) per insertion

(c) per deletion

(d) cumulative time

Figure 5.13: Synthetic shape data, 75-20-05, deletion

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

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

growing dataset size.

Varying query selectivity

We now study the robustness of GLIDE to the size of query results using the real-world ROADS and BUILDINGS data sets with 75% shuffled action ratio workload. We tune selectivity to the order of magnitude of 0.0001%, 0.001% (which is the default), 0.01%, and 0.1%. As the size of the query results grows, all indices register longer total times, as Figures 5.7e and 5.9e show with the x-axis representing selectivity. The higher the selectivity, the smaller the difference in cumulative times; this behavior was also observed in [105]. Still, under realistic query result sizes, GLIDE has a clear advantage.

Point datasets

As mentioned, to demonstrate the generality of our methods, we conducted experiments with point data sets. We use a synthetic 2D point dataset and validate GLIDE's robustness to workload distribution and dataset size. We reinstate the Quad-tree in this experiment, as it is designed for point data. Figure 5.11 shows the resulting trends, which are similar to those obtained with shape data, with the Quad-tree showing a slight improvement.

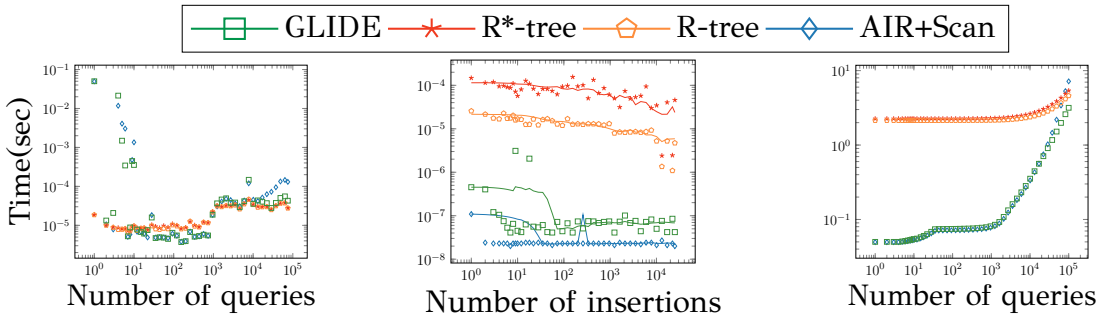
Effect of deletions

We apply expanded workloads that also include deletions on the synthetic data set. We find the object to delete in the index using its geometry; if the id of the object is given, we use it to access its geometry. To adhere to standard realistic workloads, we design a workload comprising 75% of range queries, 20% of insertions, and 5% of deletions, in shuffled order. Figure 5.13 presents our results, decoupling the query (5.13a), insertion (5.13b) and deletion times (5.13c), and also displays the progression of the overall workload time (5.13d). In all cases, GLIDE gracefully integrates deletions in its operation.

A pathological workload

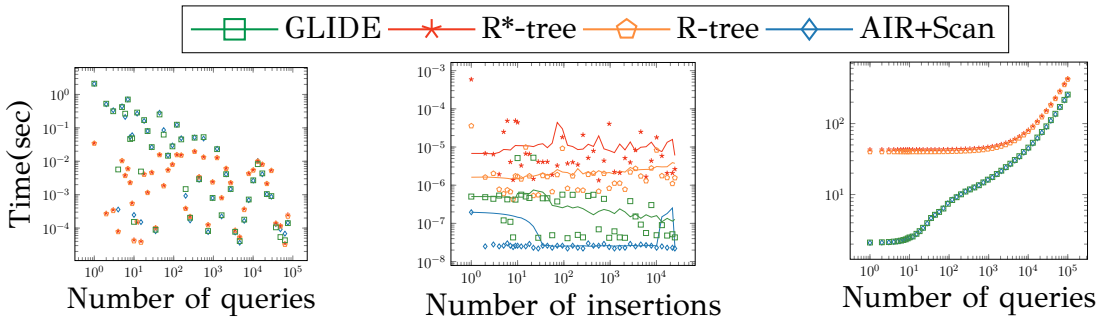
Adaptive indexing methods are vulnerable to query workloads that explore the data space in a skewed manner, especially by a *sequential* pattern [17, 21]. Here, we study

such a synthetic workload of queries ordered across a diagonal line in the 2D space intertwined with insertions; we let query extents follow a Uniform distribution in the range of [0, 0.005] to create the default target selectivity of 0.001%, and interleave them with insertions to create a 75% shuffled action-ratio workload. Figure 5.14 depicts our results. GLIDE reaches a steady-state performance with per-query time in the same order of magnitude as the R-tree in fewer than 1000 queries, while maintaining a cumulative-time advantage. This achievement is due to both the *stochastic* cracks created upon range queries and the *extra* cracks created upon insertions, as explained in Section 5.2.2. Other methods follow the trends observed in preceding experiments.



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

Figure 5.14: Uniform 2D shape data, 75-25, sequential queries



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

Figure 5.15: TLC data, 75-25, Uniform queries.

3D data

As in previous work [21], we focus on objects with spatial extent, which naturally occur in two or three dimensions. In the 3D space, we experimented with TLC. Figure 5.15 depicts our results, which corroborate GLIDE's resilience with respect to

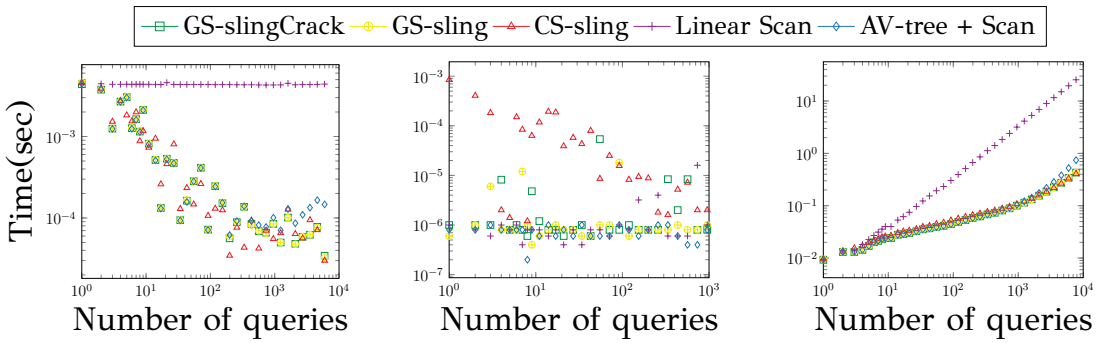
dimensionality. Owing to the small number of insertions relative to the data set's size in this case, AIR+Scan achieves performance close to GLIDE.

5.4.7 kNN workloads comparative study

We let the action ratio be 75-25 queries to insertions and investigate the results of searching for 20 nearest neighbors. As our main baselines, we include (i) a simple Linear Scan, and (ii) a naive extension of AV-tree with a simplistic array, denoted as AV-tree+Scan. We include the first as high-dimensional indexes are susceptible to the curse of dimensionality, which a brute-force algorithm may avoid. The second baseline appends inserted data to a separate log structure. When evaluating a query, we also scan this auxiliary data structure to retrieve results from inserted data. Figure 5.16 depicts these results.

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

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



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

Figure 5.16: MNIST data, 75-25, 20NN workload.

5.5 Conclusion

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

Chapter 6

Conclusions & Future Work

In conclusion, we present a summary of our significant contributions and outline potential paths for future research.

6.1 Summary of Contributions

In this dissertation, we studied adaptive indexing in various scenarios, such as data with spatial extent and high-dimensional data and explored an updating mechanism for such indexes.

In the first part of this thesis, we introduced AV-tree, to the best of our knowledge, the first adaptive index specifically designed for high-dimensional metric spaces. We proposed a novel technique for partitioning the space around query centers, organizing the data into units defined by hyperspheres, and using mediocre distance bounds that naturally adapt to the underlying data distribution. To achieve this, we developed algorithms capable of handling the most common similarity queries, including range and k-NN queries, while efficiently building and maintaining an index that adapts to the data relevant to the users' needs. Our experimental results, using datasets of various types and distance metrics, demonstrate that AV-tree outperforms both linear

scan and the current state-of-the-art adaptive index for multidimensional data, as well as a static, pivot-based index.

In the second part of this thesis, we present a comprehensive evaluation of adaptive indices. In our earlier work, we concluded that there is a significant gap in comparative studies that assess these methods on equal terms across a broad range of settings, including various data types, distributions, sizes, and workload patterns. This work aims to fill that gap by providing a detailed benchmark that thoroughly evaluates the performance, strengths, and limitations of existing multidimensional adaptive indexing methods across diverse scenarios, offering valuable insights that complement previous studies. Additionally, we compare these adaptive methods with static indices, which prove to be highly effective in certain scenarios, such as for uniformly distributed, low-dimensional data.

We also propose extensions to existing adaptive indices to address their performance limitations. Specifically, we extend the coarse-granular index, which previously only supported 1-dimensional, uniformly distributed data. Our extension suggests initially partitioning the data using a grid, where each cell is then further refined with an adaptive index. Additionally, we propose an enhancement to the AKD, introducing different heuristics for partitioning the data. This extension aims to improve performance under challenging conditions, such as irregular query patterns and varying object sizes.

Our key conclusion is that a static index, specifically the grid index, which had not been considered a competitor to adaptive indices in prior studies, performs exceptionally well on point data, offering robust performance across a variety of settings. These findings challenge the assumption that adaptive indices will always outperform static ones under reasonable workloads.

In the third and final part of this work, we introduced GLIDE, an update mechanism designed for adaptive in-memory indexes for multi-dimensional objects. While adaptive indexing has been extensively studied, existing spatial indexing methods typically cater to static datasets, where data is available all at once. No existing spatial adaptive indexing method can handle data updates interleaved with ongoing data exploration. Our proposal, GLIDE, integrates adaptive indexing with the incremental updating of spatial datasets. It allows the index to be built dynamically in response to queries while absorbing data insertions as they arrive, keeping it up-to-date. We explored the design space and developed a mechanism that incorporates updates (in-

sertions and deletions) directly into the index structure, allowing them to propagate over time. Additionally, it reorganizes the underlying data array, moving and cracking partitions as necessary. Through extensive experimentation on both synthetic and real multi-dimensional datasets, we demonstrated that GLIDE outperforms both patchwork extensions of previous adaptive indexing solutions and static indexes when responding to the same workloads.

6.2 Directions for Future Work

In this section, we outline ideas for additional research. For future work, there are several directions, on which we elaborate below:

Adaptive indexing for road networks

One promising direction for future work is to explore adaptive indexing in other spaces, such as graphs (e.g., road networks). In the context of road networks, adaptive indexing has the potential to address challenges posed by the complex network structure and the diverse query patterns. A promising avenue for exploration is to potentially exploit the graph structure to gain insights into how to expand the index. The graph structure can provide valuable clues about regions that may benefit from indexing expansion. For example, the presence of a strongly connected component could suggest that certain areas are more likely to receive frequent queries, indicating that the index should be expanded in those regions. Identifying such a component can also help in selecting landmark nodes, as all nodes within the component are mutually accessible. This can facilitate the creation of a hierarchical structure (or partitioning) within the graph, reducing the number of distance computations required and improving query efficiency. By leveraging these structural patterns, the adaptive indexing approach could dynamically adjust to query needs based on the network's topology, making it more efficient and responsive to evolving query patterns.

Another approach could be to develop a novel adaptive indexing strategy specifically for graph data, or to adapt existing graph-based indexing techniques, such as hub labeling, to integrate with the adaptive indexing scheme. Furthermore, the spatial and topological properties of the nodes and edges in a graph leave room for experimentation with various partitioning techniques, potentially enhancing the indexing strategy's ability to process different types of queries.

The Dijkstra algorithm is widely used for finding the shortest path in a weighted graph with non-negative edge weights. In each iteration, the algorithm computes the shortest path from the source node to a new node and adds it to the set of visited nodes. However, the values computed during each iteration are typically discarded after completion. An interesting direction for future work is to explore the potential benefits of caching and reusing these values to speed up the process for subsequent queries, effectively building an adaptive index for shortest path queries. For example, cached values could be used to guide the search for the shortest path, helping the algorithm focus on areas of the graph that are more likely to lead to the desired outcome. This approach could minimize the need for recalculating distances along paths already explored. Additionally, cached values could inform the priority queue in the algorithm, enabling it to prioritize nodes based on their computed distance from previous iterations. This could result in faster convergence and fewer redundant iterations. Finally, drawing inspiration from database cracking techniques, we could leverage the shortest path tree generated during each iteration to partition the graph into meaningful segments, such as dividing the network into closer and farther components based on their distance from the query source. This partitioning could enhance the efficiency of subsequent searches by narrowing the search space.

Multiway AV-tree

The AV-tree has demonstrated promising results in efficiently indexing and querying high-dimensional data in metric spaces. However, to further enhance its scalability and performance, two potential extensions warrant exploration: extending to an m-ary tree structure and implementing re-balancing techniques. The first extension involves partitioning the data space into m disjoint subspaces, which, unlike the current binary space partitioning, would divide the space around each query into layers based on multiple distance bounds. A promising avenue for investigation is whether this m-ary tree extension offers greater benefits when applied to data structured as graph data, as discussed earlier. The second extension focuses on re-balancing the tree, which involves adjusting the order of the nodes to optimize the index structure and improve performance for subsequent queries.

Bibliography

[1] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 2007, pp. 68–78. [Online]. Available: http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[2] S. Idreos, "Database cracking: Towards auto-tuning database kernels," Ph.D. dissertation, CWI, 2010.

[3] P. A. Boncz and M. L. Kersten, "MIL primitives for querying a fragmented world," *VLDB J.*, vol. 8, no. 2, pp. 101–119, 1999. [Online]. Available: https://doi.org/10.1007/s007780050076

[4] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-store: A column-oriented DBMS," in *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P. Larson, and B. C. Ooi, Eds. ACM, 2005, pp. 553–564. [Online]. Available: http://www.vldb.org/archives/website/2005/program/paper/thu/p553-stonebraker.pdf

[5] S. Idreos, M. L. Kersten, and S. Manegold, "Self-organizing tuple reconstruction in column-stores," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, Eds. ACM, 2009, pp. 297–308. [Online]. Available: https://doi.org/10.1145/1559845.1559878

[6] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, no. 7, p. 321, 1961. [Online]. Available: https://doi.org/10.1145/366622.366644

[7] M. Liu, D. Li, Q. Chen, J. Zhou, K. Meng, and S. Zhang, "Sensor information retrieval from internet of things: Representation and indexing," *IEEE Access*, vol. 6, pp. 36 509–36 521, 2018. [Online]. Available: https://doi.org/10.1109/ACCESS.2018.2849865

[8] Y. Fathy, P. M. Barnaghi, and R. Tafazolli, "Large-scale indexing, discovery, and ranking for the internet of things (iot)," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 29:1–29:53, 2018. [Online]. Available: https://doi.org/10.1145/3154525

[9] Q.-T. Doan, A. S. M. Kayes, W. Rahayu, and K. Nguyen, "A framework for iot streaming data indexing and query optimization," *IEEE Sensors Journal*, vol. 22, no. 14, pp. 14 436–14 447, 2022.

[10] N. Chaudhry, M. M. Yousaf, and M. T. Khan, "Indexing of real time geospatial data by iot enabled devices: Opportunities, challenges and design considerations," *J. Ambient Intell. Smart Environ.*, vol. 12, no. 4, pp. 281–312, 2020. [Online]. Available: https://doi.org/10.3233/AIS-200565

[11] A. Nanopoulos, Y. Theodoridis, and Y. Manolopoulos, "C^{2}p: Clustering based on closest pairs," in *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, Eds. Morgan Kaufmann, 2001, pp. 331–340. [Online]. Available: http://www.vldb.org/conf/2001/P331.pdf

[12] C. C. Aggarwal and P. S. Yu, "Outlier detection for high dimensional data," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, S. Mehrotra and T. K. Sellis, Eds. ACM, 2001, pp. 37–46. [Online]. Available: https://doi.org/10.1145/375663.375668

[13] Y. Rui, T. S. Huang, and S. Chang, "Image retrieval: Current techniques, promising directions, and open issues," *J. Vis. Commun. Image Represent.*, vol. 10, no. 1, pp. 39–62, 1999. [Online]. Available: https://doi.org/10.1006/jvci.1999.0413

[14] R. Connor, "A tale of four metrics," in *Similarity Search and Applications - 9th International Conference, SISAP 2016, Tokyo, Japan, October 24-26, 2016.*

Proceedings, ser. Lecture Notes in Computer Science, L. Amsaleg, M. E. Houle, and E. Schubert, Eds., vol. 9939, 2016, pp. 210–217. [Online]. Available: https://doi.org/10.1007/978-3-319-46759-7_16

[15] T. Abeywickrama, M. A. Cheema, and D. Taniar, "k-nearest neighbors on road networks: A journey in experimentation and in-memory implementation," *Proc. VLDB Endow.*, vol. 9, no. 6, pp. 492–503, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol9/p492-abeywickrama.pdf

[16] H. Li, T. N. Chan, M. L. Yiu, and N. Mamoulis, "FEXIPRO: fast and exact inner product retrieval in recommender systems," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds. ACM, 2017, pp. 835–850. [Online]. Available: https://doi.org/10.1145/3035918.3064009

[17] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap, "Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores," *Proc. VLDB Endow.*, vol. 5, no. 6, pp. 502–513, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p502_felixhalim_vldb2012.pdf

[18] M. A. Nerone, P. Holanda, E. C. de Almeida, and S. Manegold, "Multidimensional adaptive & progressive indexes," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 624–635. [Online]. Available: https://doi.org/10.1109/ICDE51399.2021.00060

[19] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki, "QUASII: query-aware spatial incremental index," in *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, M. H. Böhlen, R. Pichler, N. May, E. Rahm, S. Wu, and K. Hose, Eds. OpenProceedings.org, 2018, pp. 325–336. [Online]. Available: https://doi.org/10.5441/002/edbt.2018.29

[20] P. Holanda, M. Nerone, E. C. de Almeida, and S. Manegold, "Cracking kd-tree: The first multidimensional adaptive indexing (position paper)," in *Proceedings of the 7th International Conference on Data Science, Technology and*

Applications, *DATA 2018*, *Porto*, *Portugal*, *July 26-28*, *2018*, J. Bernardino and C. Quix, Eds. SciTePress, 2018, pp. 393–399. [Online]. Available: https://doi.org/10.5220/0006944203930399

[21] F. Zardbani, N. Mamoulis, S. Idreos, and P. Karras, "Adaptive indexing of objects with spatial extent," *Proc. VLDB Endow.*, vol. 16, no. 9, pp. 2248–2260, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p2248-karras.pdf

[22] K. Lampropoulos, F. Zardbani, N. Mamoulis, and P. Karras, "Adaptive indexing in high-dimensional metric spaces," *Proc. VLDB Endow.*, vol. 16, no. 10, pp. 2525–2537, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p2525-mamoulis.pdf

[23] P. Holanda, S. Manegold, H. Mühleisen, and M. Raasveldt, "Progressive indexes: Indexing for interactive data analysis," *Proc. VLDB Endow.*, vol. 12, no. 13, pp. 2366–2378, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2366-holanda.pdf

[24] M. L. Kersten and S. Manegold, "Cracking the database store," in *Second Biennial Conference on Innovative Data Systems Research*, *CIDR 2005*, *Asilomar*, *CA*, *USA*, *January 4-7*, *2005*, *Online Proceedings*. www.cidrdb.org, 2005, pp. 213–224. [Online]. Available: http://cidrdb.org/cidr2005/papers/P18.pdf

[25] F. M. Schuhknecht, A. Jindal, and J. Dittrich, "The uncracked pieces in database cracking," *Proc. VLDB Endow.*, vol. 7, no. 2, pp. 97–108, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol7/p97-schuhknecht.pdf

[26] ——, "An experimental evaluation and analysis of database cracking," *VLDB J.*, vol. 25, no. 1, pp. 27–52, 2016. [Online]. Available: https://doi.org/10.1007/s00778-015-0397-y

[27] A. H. Jensen, F. Lauridsen, F. Zardbani, S. Idreos, and P. Karras, "Revisiting multidimensional adaptive indexing [experiment & analysis]," in *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021*, *Nicosia*, *Cyprus*, *March 23 - 26*, *2021*, Y. Velegrakis, D. Zeinalipour-Yazti, P. K. Chrysanthis, and F. Guerra, Eds. OpenProceedings.org, 2021, pp. 469–474. [Online]. Available: https://doi.org/10.5441/002/edbt.2021.53

[28] P. Holanda, M. Nerone, E. C. de Almeida, and S. Manegold, "Cracking kd-tree: The first multidimensional adaptive indexing (position paper)," in *Proceedings of the 7th International Conference on Data Science, Technology and Applications, DATA 2018, Porto, Portugal, July 26-28, 2018*, J. Bernardino and C. Quix, Eds. SciTePress, 2018, pp. 393–399. [Online]. Available: https://doi.org/10.5220/0006944203930399

[29] F. Zardbani, P. Afshani, and P. Karras, "Revisiting the theory and practice of database cracking," in *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, Eds. OpenProceedings.org, 2020, pp. 415–418. [Online]. Available: https://doi.org/10.5441/002/edbt.2020.46

[30] S. Idreos, M. L. Kersten, and S. Manegold, "Updating a cracked database," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, C. Y. Chan, B. C. Ooi, and A. Zhou, Eds. ACM, 2007, pp. 413–424. [Online]. Available: https://doi.org/10.1145/1247480.1247527

[31] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, B. Yormark, Ed. ACM Press, 1984, pp. 47–57. [Online]. Available: https://doi.org/10.1145/602259.602266

[32] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," in *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix)*, E. F. Codd, Ed. ACM, 1970, pp. 107–141. [Online]. Available: https://doi.org/10.1145/1734663.1734671

[33] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, H. Garcia-Molina and H. V. Jagadish, Eds. ACM Press, 1990, pp. 322–331. [Online]. Available: https://doi.org/10.1145/93597.98741

[34] R. A. Finkel and J. L. Bentley, "Quad trees: A data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, pp. 1–9, 1974. [Online]. Available: https://doi.org/10.1007/BF00288933

[35] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975. [Online]. Available: https://doi.org/10.1145/361002.361007

[36] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch, "Main memory evaluation of monitoring queries over moving objects," *Distributed Parallel Databases*, vol. 15, no. 2, pp. 117–135, 2004. [Online]. Available: https://doi.org/10.1023/B:DAPD.0000013068.25976.88

[37] D. Sidlauskas, S. Saltenis, C. W. Christiansen, J. M. Johansen, and D. Saulys, "Trees or grids?: indexing moving objects in main memory," in *17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November 4-6, 2009, Seattle, Washington, USA, Proceedings*, D. Agrawal, W. G. Aref, C. Lu, M. F. Mokbel, P. Scheuermann, C. Shahabi, and O. Wolfson, Eds. ACM, 2009, pp. 236–245. [Online]. Available: https://doi.org/10.1145/1653771.1653805

[38] X. Yu, K. Q. Pu, and N. Koudas, "Monitoring k-nearest neighbor queries over moving objects," in *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, K. Aberer, M. J. Franklin, and S. Nishio, Eds. IEEE Computer Society, 2005, pp. 631–642. [Online]. Available: https://doi.org/10.1109/ICDE.2005.92

[39] D. Tsitsigkos, P. Bouros, K. Lampropoulos, N. Mamoulis, and M. Terrovitis, "Two-layer space-oriented partitioning for non-point data," *CoRR*, vol. abs/2307.09256, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09256

[40] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is "nearest neighbor" meaningful?" in *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, ser. Lecture Notes in Computer Science, C. Beeri and P. Buneman, Eds., vol. 1540. Springer, 1999, pp. 217–235. [Online]. Available: https://doi.org/10.1007/3-540-49257-7_15

[41] L. Chen, Y. Gao, X. Song, Z. Li, Y. Zhu, X. Miao, and C. S. Jensen, "Indexing metric spaces for exact similarity search," *ACM Comput. Surv.*, vol. 55, no. 6, pp. 128:1–128:39, 2023. [Online]. Available: https://doi.org/10.1145/3534963

[42] L. Chen, Y. Gao, B. Zheng, C. S. Jensen, H. Yang, and K. Yang, "Pivot-based metric indexing," *Proc. VLDB Endow.*, vol. 10, no. 10, pp. 1058–1069, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1058-gao.pdf

[43] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. ACM, 2013, pp. 349–360. [Online]. Available: https://doi.org/10.1145/2463676.2465315

[44] N. S. Detlefsen, S. Hauberg, and W. Boomsma, "Learning meaningful representations of protein sequences," *Nature Communications*, vol. 13, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:227305297

[45] S. Li, K. Li, J. Yang, Y. Liu, W. Han, and Y. Luo, "Research on the local regional similarity of automatic fingerprint identification system fingerprints based on close non-matches in a ten million people database – taking the central region of whorl as an example," *Journal of Forensic Sciences*, vol. 68, no. 2, pp. 488–499, 2023.

[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[47] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, Eds. Morgan Kaufmann, 1999, pp. 518–529. [Online]. Available: http://www.vldb.org/conf/1999/P49.pdf

[48] Y. Tian, X. Zhao, and X. Zhou, "DB-LSH: locality-sensitive hashing with query-based dynamic bucketing," in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022, pp. 2250–2262. [Online]. Available: https://doi.org/10.1109/ICDE53745.2022.00214

[49] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, pp. 824–836, 2020. [Online]. Available: https://doi.org/10.1109/TPAMI.2018.2889473

[50] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *Proc. VLDB Endow.*, vol. 12, no. 5, pp. 461–474, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p461-fu.pdf

[51] D. S. Hochbaum and D. B. Shmoys, "A best possible heuristic for the *k*-center problem," *Math. Oper. Res.*, vol. 10, no. 2, pp. 180–184, 1985. [Online]. Available: https://doi.org/10.1287/moor.10.2.180

[52] G. Navarro, "Searching in metric spaces by spatial approximation," *VLDB J.*, vol. 11, no. 1, pp. 28–46, 2002. [Online]. Available: https://doi.org/10.1007/s007780200060

[53] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, V. Ramachandran, Ed. ACM/SIAM, 1993, pp. 311–321. [Online]. Available: http://dl.acm.org/citation.cfm?id=313559.313789

[54] T. Bozkaya and Z. M. Özsoyoglu, "Indexing large metric spaces for similarity search queries," *ACM Trans. Database Syst.*, vol. 24, no. 3, pp. 361–404, 1999. [Online]. Available: https://doi.org/10.1145/328939.328959

[55] A. W. Fu, P. M. Chan, Y. Cheung, and Y. S. Moon, "Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances," *VLDB J.*, vol. 9, no. 2, pp. 154–173, 2000. [Online]. Available: https://doi.org/10.1007/PL00010672

[56] H. V. Jagadish, B. C. Ooi, K. Tan, C. Yu, and R. Zhang, "idistance: An adaptive b^+-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 364–397, 2005. [Online]. Available: https://doi.org/10.1145/1071610.1071612

[57] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds. Morgan Kaufmann, 1997, pp. 426–435. [Online]. Available: http://www.vldb.org/conf/1997/P426.PDF

[58] T. Skopal, J. Pokorný, and V. Snásel, "Pm-tree: Pivoting metric tree for similarity search in multimedia databases," in *Advances in Databases and Information Systems, 8th East European Conference, ADBIS 2004, Budapest, Hungary, September 22-25, 2004, Local Proceeding*, 2004. [Online]. Available: http://www.sztaki.hu/conferences/ADBIS/9-Skopal.pdf

[59] R. M. Karp, R. Motwani, and P. Raghavan, "Deferred data structuring," *SIAM J. Comput.*, vol. 17, no. 5, pp. 883–902, 1988. [Online]. Available: https://doi.org/10.1137/0217055

[60] P. Karras, A. Nikitin, M. Saad, R. Bhatt, D. Antyukhov, and S. Idreos, "Adaptive indexing over encrypted numeric data," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 171–183. [Online]. Available: https://doi.org/10.1145/2882903.2882932

[61] G. Graefe and H. A. Kuno, "Self-selecting, self-tuning, incrementally optimized indexes," in *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, ser. ACM International Conference Proceeding Series, I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Léger, F. Naumann, A. Ailamaki, and F. Özcan, Eds., vol. 426. ACM, 2010, pp. 371–381. [Online]. Available: https://doi.org/10.1145/1739041.1739087

[62] P. Holanda and S. Manegold, "Progressive mergesort: Merging batches of appends into progressive indexes," in *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, Y. Velegrakis, D. Zeinalipour-Yazti, P. K. Chrysanthis, and F. Guerra, Eds. OpenProceedings.org, 2021, pp. 481–486. [Online]. Available: https://doi.org/10.5441/002/edbt.2021.55

[63] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe, "Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 585–597, 2011. [Online]. Available: http://www.vldb.org/pvldb/vol4/p586-idreos.pdf

[64] S. Idreos, M. L. Kersten, and S. Manegold, "Self-organizing tuple reconstruction in column-stores," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, Eds. ACM, 2009, pp. 297–308. [Online]. Available: https://doi.org/10.1145/1559845.1559878

[65] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 489–504. [Online]. Available: https://doi.org/10.1145/3183713.3196909

[66] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 1189–1206. [Online]. Available: https://doi.org/10.1145/3299869.3319860

[67] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. B. Lomet, and T. Kraska, "ALEX: an updatable adaptive learned index," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan,

A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 969–984. [Online]. Available: https://doi.org/10.1145/3318464.3389711

[68] P. Ferragina and G. Vinciguerra, "The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1162–1175, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p1162-ferragina.pdf

[69] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 985–1000. [Online]. Available: https://doi.org/10.1145/3318464.3380579

[70] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "Radixspline: a single-pass learned index," in *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, R. Bordawekar, O. Shmueli, N. Tatbul, and T. K. Ho, Eds. ACM, 2020, pp. 5:1–5:5. [Online]. Available: https://doi.org/10.1145/3401071.3401659

[71] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: A learned multi-dimensional index for correlated data and skewed workloads," *Proc. VLDB Endow.*, vol. 14, no. 2, pp. 74–86, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol14/p74-ding.pdf

[72] A. Crotty, "Hist-tree: Those who ignore it are doomed to learn," in *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. [Online]. Available: http://cidrdb.org/cidr2021/papers/cidr2021_paper20.pdf

[73] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST: fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA,*

June 6-10, 2010, A. K. Elmagarmid and D. Agrawal, Eds. ACM, 2010, pp. 339–350. [Online]. Available: https://doi.org/10.1145/1807167.1807206

[74] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan, "Sagedb: A learned database system," in *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. [Online]. Available: http://cidrdb.org/cidr2019/papers/p117-kraska-cidr19.pdf

[75] P. Ferragina, F. Lillo, and G. Vinciguerra, "Why are learned indexes so effective?" in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 3123–3132. [Online]. Available: http://proceedings.mlr.press/v119/ferragina20a.html

[76] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, C. S. Jensen, C. M. Jermaine, and X. Zhou, Eds. IEEE Computer Society, 2013, pp. 38–49. [Online]. Available: https://doi.org/10.1109/ICDE.2013.6544812

[77] M. Mitzenmacher, "A model for learned bloom filters and optimizing by sandwiching," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 462–471. [Online]. Available: https://proceedings.neurips.cc/paper/2018/hash/0f49c89d1e7298bb9930789c8ed59d48-Abstract.html

[78] A. Kipf, D. Horn, P. Pfeil, R. Marcus, and T. Kraska, "LSI: a learned secondary index structure," in *aiDM '22: Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, Philadelphia, Pennsylvania, USA, 17 June 2022*, R. Bordawekar, O. Shmueli, Y. Amsterdamer, D. Firmani, and R. Marcus, Eds. ACM, 2022, pp. 4:1–4:5. [Online]. Available: https://doi.org/10.1145/3533702.3534912

[79] K. Vaidya, E. Knorr, M. Mitzenmacher, and T. Kraska, "Partitioned learned bloom filters," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=6BRLOfrMhW

[80] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "SOSD: A benchmark for learned indexes," *CoRR*, vol. abs/1911.13014, 2019. [Online]. Available: http://arxiv.org/abs/1911.13014

[81] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska, "Benchmarking learned indexes," *Proc. VLDB Endow.*, vol. 14, no. 1, pp. 1–13, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1-marcus.pdf

[82] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, "Are updatable learned indexes ready?" *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 3004–3017, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p3004-wongkham.pdf

[83] A. Al-Mamun, H. Wu, Q. He, J. Wang, and W. G. Aref, "A survey of learned indexes for the multi-dimensional space," 2024. [Online]. Available: https://arxiv.org/abs/2403.06456

[84] Y. Chronis, Y. Wang, Y. Gan, S. Abu-El-Haija, C. Lin, C. Binnig, and F. Özcan, "Cardbench: A benchmark for learned cardinality estimation in relational databases," *CoRR*, vol. abs/2408.16170, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.16170

[85] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper, "Learned cardinalities: Estimating correlated joins with deep learning," in *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. [Online]. Available: http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[86] A. Kipf, D. Vorona, J. Müller, T. Kipf, B. Radke, V. Leis, P. A. Boncz, T. Neumann, and A. Kemper, "Estimating cardinalities with deep sketches," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30*

- *July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 1937–1940. [Online]. Available: https://doi.org/10.1145/3299869.3320218

[87] L. Deng, "The MNIST database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, 2012. [Online]. Available: https://doi.org/10.1109/MSP.2012.2211477

[88] W. Li, J. Mao, Y. Zhang, and S. Cui, "Fast similarity search via optimal sparse lifting," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 176–184. [Online]. Available: https://proceedings.neurips.cc/paper/2018/hash/a8baa56554f96369ab93e4f3bb068c22-Abstract.html

[89] O. Jafari, P. Nagarkar, and J. Montaño, "Improving locality sensitive hashing by efficiently finding projected nearest neighbors," in *Similarity Search and Applications - 13th International Conference, SISAP 2020, Copenhagen, Denmark, September 30 - October 2, 2020, Proceedings*, ser. Lecture Notes in Computer Science, S. Satoh, L. Vadicamo, A. Zimek, F. Carrara, I. Bartolini, M. Aumüller, B. Þ. Jónsson, and R. Pagh, Eds., vol. 12440. Springer, 2020, pp. 323–337. [Online]. Available: https://doi.org/10.1007/978-3-030-60936-8_25

[90] D. Alvarez-Melis and N. Fusi, "Geometric dataset distances via optimal transport," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/f52a7b2610fb4d3f74b4106fb80b233d-Abstract.html

[91] L. McInnes, J. Healy, N. Saul, and L. Großberger, "UMAP: uniform manifold approximation and projection," *J. Open Source Softw.*, vol. 3, no. 29, p. 861, 2018. [Online]. Available: https://doi.org/10.21105/joss.00861

[92] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: https://dl.acm.org/doi/10.5555/1953048.2078195

[93] A. Tsitsulin, M. Munkhoeva, D. Mottin, P. Karras, A. M. Bronstein, I. V. Oseledets, and E. Müller, "The shape of data: Intrinsic distance for data distributions," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=HyebplHYwB

[94] P. Karras and N. Mamoulis, "Hierarchical synopses with optimal error guarantees," *ACM Trans. Database Syst.*, vol. 33, no. 3, pp. 18:1–18:53, 2008. [Online]. Available: https://doi.org/10.1145/1386118.1386124

[95] Boost, "Boost C++ Libraries," http://www.boost.org/, 2015, last accessed 2025-01-10.

[96] E. Stefanakis, Y. Theodoridis, T. K. Sellis, and Y. Lee, "Point representation of spatial objects and query window extension: A new technique for spatial access methods," *Int. J. Geogr. Inf. Sci.*, vol. 11, no. 6, pp. 529–554, 1997. [Online]. Available: https://doi.org/10.1080/136588197242185

[97] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, Eds. IEEE Computer Society, 2015, pp. 1352–1363. [Online]. Available: https://doi.org/10.1109/ICDE.2015.7113382

[98] D. Kwon, S. Lee, and S. Lee, "Indexing the current positions of moving objects using the lazy update r-tree," in *Proceedings of the Third International Conference on Mobile Data Management (MDM 2002), Singapore, January 8-11, 2002*. IEEE Computer Society, 2002, pp. 113–120. [Online]. Available: https://doi.org/10.1109/MDM.2002.994387

[99] P. Katiyar, T. Vu, A. Eldawy, S. Migliorini, and A. Belussi, "Spiderweb: A spatial data generator on the web," in *SIGSPATIAL '20: 28th International*

Conference on Advances in Geographic Information Systems, Seattle, WA, USA, November 3-6, 2020, C. Lu, F. Wang, G. Trajcevski, Y. Huang, S. D. Newsam, and L. Xiong, Eds. ACM, 2020, pp. 465–468. [Online]. Available: https://doi.org/10.1145/3397536.3422351

[100] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, Eds. IEEE Computer Society, 2015, pp. 1352–1363. [Online]. Available: https://doi.org/10.1109/ICDE.2015.7113382

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

[102] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, Eds. ACM, 2010, pp. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[103] S. T. Leutenegger, J. M. Edgington, and M. A. López, "STR: A simple and efficient algorithm for r-tree packing," in *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, W. A. Gray and P. Larson, Eds. IEEE Computer Society, 1997, pp. 497–506. [Online]. Available: https://doi.org/10.1109/ICDE.1997.582015

[104] L. Xing, E. Lee, T. An, B. Chu, A. Mahmood, A. M. Aly, J. Wang, and W. G. Aref, "An experimental evaluation and investigation of waves of misery in r-trees," *Proc. VLDB Endow.*, vol. 15, no. 3, pp. 478–490, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol15/p478-aref.pdf

[105] T. Gu, K. Feng, G. Cong, C. Long, Z. Wang, and S. Wang, "The rlr-tree: A reinforcement learning based r-tree for spatial data," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 63:1–63:26, 2023. [Online]. Available: https://doi.org/10.1145/3588917

Author's Publications

- K. Lampropoulos, F. Zardbani, N. Mamoulis, P. Karras: Benchmarking adaptive indexes for spatial data. **VLDB** 2025 (under review)
- F. Zardbani, K. Lampropoulos, N. Mamoulis, P. Karras: Updating an Adaptive Spatial Index. **ICDE** 2025
- A. Michalopoulos, K. Lampropoulos, G. Kelantonakis, C. Zeginis, K. Magoutis, and N. Mamoulis: Similarity Search based on Geo-footprints. **EDBT** 2024
- K. Lampropoulos, F. Zardbani, P. Karras, N. Mamoulis: Adaptive Indexing in High-Dimensional Metric Spaces. **VLDB** 2023
- D. Tsitsigkos, K. Lampropoulos, P. Bouros, N. Mamoulis, M. Terrovitis: A Two-level Partitioning for Non-point Spatial Data. **ICDE** 2021
- P. Bouros, K. Lampropoulos, D. Tsitsigkos, N. Mamoulis, M. Terrovitis: Band Joins for Interval Data. **EDBT** 2020

Short Biography

Konstantinos Lampropoulos was born in 1993. He received his diploma in Computer Science and Engineering from the Department of Computer Science and Engineering at the University of Ioannina in 2019. During his PhD studies, he visited the Department of Computer Science at Aarhus University in Denmark, where he worked under the supervision of Prof. Panagiotis Karras. His research interests lie in data management, with a particular focus on in-memory databases, and (adaptive) indexing of complex data types, such as temporal, spatial, and high-dimensional data.