IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 19 May 2025, accepted 9 June 2025, date of publication 19 June 2025, date of current version 26 June 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3581403

RESEARCH ARTICLE

Efficient Exploratory Searches With an Incremental R-Tree Under Oscillating Workloads

MUSTAFA UTKU KALAY, (Member, IEEE)
Computer Engineering Department, Yildiz Technical University, 34220 İstanbul, Türkiye
e-mail: ukalay@yildiz.edu.tr

This work was supported by Yildiz Technical University Scientific Research Projects Coordinatorship under Grant FKD-2024-6620.

ABSTRACT Traditional spatial index structures face challenges in efficiently managing dynamic and irregular workloads, which are common in real-time spatial data analysis. In particular, oscillating workloads can significantly hinder data access performance due to the continuous influx of spatial records in varying volume and frequency. To address these challenges, this paper presents an Incremental R-tree index structure optimized for real-time storage and querying of large-scale spatial datasets under dynamic conditions. The proposed method employs a two-stage overflow mechanism, where newly arriving data batches are first distributed into a grid-based structure and then selectively integrated into the tree leaves based on query activity. This approach delays full indexing until it becomes necessary, thereby improving responsiveness and avoiding unnecessary tree restructuring. Furthermore, the Incremental R-tree adaptively incorporates principles from both STR-tree and R*-tree to enhance indexing efficiency. Experimental results demonstrate that the Incremental R-tree significantly reduces cumulative query response latency compared to traditional R*-tree and STR-tree structures under oscillating workloads, and consistently outperforms the widely used GiST-based R-tree implementation in PostGIS.

INDEX TERMS Workload-optimized spatial indexes, R-tree, exploratory search, query processing performance.

I. INTRODUCTION

Nowadays, rapidly growing datasets are subjected to varying patterns of workloads. These patterns may include periodic read and write queries, as well as bursty inserts (i.e., irregular, high-volume data arrivals in short intervals) caused by sudden events. Moreover, both the range of data being inserted or updated and the regions of data being queried often vary based on popularity or instantaneous events. Among the several examples of such dynamic environments, spatial exploratory data analysis systems enable real-time exploration and analysis of spatially-referenced data, such as geographic locations, shapes, and spatial relationships. These systems have wide applications in domains such as disaster management, crime analysis, and transportation management. Integrating incoming data and adapting to changing demands without interrupting ongoing analysis pose significant challenges for these systems. As an example, Intelligent Traffic Management Systems (ITMS) utilize real-time traffic data to optimize traffic flow, reduce congestion, and provide navigation guidance to drivers. However, a sudden downpour can cause abrupt changes in traffic patterns, hindering the system's ability to process new data quickly due to the inability of current spatial systems to adapt to these changes.

Generally, spatial indexes, such as R-trees [1], are used to organize large-scale spatial data sets, enabling efficient search and retrieval. While not worst-case optimal like B-trees, R-trees are a widely used data structure known for being read-optimized. This efficiency stems from R-tree's effort to organize data near-perfectly, but this very characteristic hinders the tree from being write-optimized. As it is more common for data to emerge in bulk rather than individually, specifically in today's batch-oriented processing systems, the improvement of write efficiency in data structures has become increasingly important. Both the

The associate editor coordinating the review of this manuscript and approving it for publication was Barbara Guidi.

© 2025 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

initial bulk construction of the tree and the bulk insertions of newly incoming datasets are preferable options compared to dynamic updates (i.e., individual updates) [18]. This is because bulk updates minimize disk I/O by consolidating changes and optimize the use of memory and CPU caches. When data is processed in batches, it is usually preprocessed (i.e., sorted or clustered) to enhance its integration into the index. There have been many studies that focus on the efficient initial construction of the R-tree with bulk data [2], [3], [4], [5], [6], [7], [8], [9]. Recent studies predominantly focused on making the construction process workload-sensitive in different data processing platforms [10], [11], [12], [13], [14], [15], [16], [17].

Alongside the efficient construction of the initial tree, the effective integration of newly emerging data batches into the structure presents a significant challenge. Various studies in the literature offer solutions related to this issue [18], [19], [20], [21], [22]. Furthermore, it is common for new data batches to continuously appear at regular or irregular intervals in today's workloads. As a result, oscillating workloads exhibit periods of both read-heavy and write-heavy behavior. This read-write oscillation can occur in regular cycles or manifest irregularly due to unexpected events. For example, in data warehouses, workloads tend to be read-intensive during the day and write-intensive at night [23]. Conversely, social media platforms often experience write-intensive workloads during the day and more subdued, read-heavy behavior at night [24]. This mixed behavior has also been suggested in new performance testing environments, such as Hybrid Transactional and Analytical Processing systems (HTAP) [25]. Additionally, the irregularity of both queried and newly added data within potential data ranges further increases the existing dynamics. This popular type of continuously changing workload has been used as a performance benchmark in many studies [23], [24], [25], [26].

Efficient handling of continuous bulk insertions under oscillating spatial workloads remains a significant challenge, one that this study addresses by building upon prior observations and established findings. The main objective is to support progressive data ingestion while maintaining stable and efficient query performance.

Fully integrating all incoming spatial data into an existing R-tree—even using state-of-the-art bulk insertion strategies such as those proposed in [18], [19], [20], [21], and [22]—can be both costly and unnecessary, particularly in regions that are rarely queried. Indexing data from such areas unnecessarily enlarges the tree, which in turn degrades query efficiency in more frequently accessed regions due to increased overlap and tree depth. This issue becomes especially critical when incoming data arrives in large volumes and in short bursts.

As a result, alternative integration strategies must be explored—strategies that not only avoid indexing unused data but also minimize expensive random-access operations during both data ingestion and query evaluation. To meet these requirements, the proposed method is designed around two core principles:

(1) Incremental indexing—only a subset of the incoming data should be indexed, based on query relevance; and

(2) Minimization of random access, particularly by keeping new records temporarily in overflow structures rather than immediately inserting them into the tree.

Inspired by these design principles, the method first distributes the newly arriving records into buckets governed by a static fixed grid, and then forwards only the query-relevant portions to the R-tree leaves. This forwarding is performed in bulk, rather than record-by-record, thereby reducing insertion overhead and improving access locality. This selective ingestion enables coarse partitioning based on spatial proximity and supports efficient sequential access to these subsets at the leaves via the seed R-tree during read-intensive phases.

To fully integrate overflowed data at the leaves, a holistic and cost-based strategy is introduced. Subsets are gradually incorporated into the tree structure based on their access frequency, monitored across both the current and previous read-intensive cycles. Additionally, the cost model takes into account spatial criteria to improve the accuracy of selecting which partitions should be integrated next.

To validate these principles and realize them in practice, a novel index structure—called the Incremental R-tree—was implemented. The Incremental R-tree structure has been implemented through targeted modifications to the libspatialindex C++ library (available at https://libspatialindex.org). To ensure high-throughput bulk insertions while maintaining read efficiency, the proposed structure was evaluated under an oscillating workload composed of alternating read-heavy and write-heavy phases. The exploration simulation framework developed in this study focuses on query performance across varying access patterns—such as surface-level and deep exploration—applied to both synthetic and real-world datasets.

As a first step, the behavior of the Incremental R-tree was examined under a synthetic workload, which exhibited oscillatory but relatively regular patterns.

To further assess its performance under highly dynamic and irregular workloads, a real-world dataset was constructed by periodically collecting passenger pickup locations of NYC yellow taxis throughout the day. In these real-world scenarios, the resilience of the Incremental R-tree was demonstrated by its ability to retain non-queried regions within Grid overflow buckets, and to defer the integration of sparsely queried regions by temporarily holding them within leaf-level overflow buffers until integration becomes necessary. This adaptive behavior ensured that unnecessary indexing was avoided in inactive areas, while query-intensive regions were incrementally integrated based on demand.

Experimental results show that the Incremental R-tree significantly outperforms classical structures such as the R*-tree, STR-tree, and the widely adopted GiST-based R-tree in PostGIS (https://postgis.net/), across diverse data ingestion and query workload scenarios. While similar oscillating

workloads have been explored in distributed big spatial data contexts (e.g., [11]), to the best of our knowledge, no prior study has addressed centralized R-tree structures under such highly dynamic spatial conditions.

The contributions of this paper can be summarized as follows:

- A more robust solution for highly dynamic, oscillating workloads is provided by extending the libspatialindex C++ library with traditional R-tree algorithms (STR-tree and R-tree*), resulting in the Incremental R-tree.
- A strategy of adding new data batches exclusively to leaf nodes is adopted, thereby preventing unnecessary tree growth due to the incoming data to infrequently accessed regions.
- To enhance overall efficiency, a cost model is introduced, utilizing some spatial criteria and access frequency to improve the selection of partitions for integration.
- The proposed method integrates three core mechanisms:
  (i) Distributing bursty datasets into static Grid File buckets,
  (ii) Selectively forwarding query-relevant records to overflow buckets at the leaf level, and
  (iii) Incrementally integrating these overflow buckets into the tree based on query activity.
- An extensive experimental setup using oscillating workloads was designed to evaluate cumulative performance. Results indicate that the Incremental R-tree consistently outperforms traditional structures—including the R*-tree, STR-tree, and the GiST-R-tree used in PostGIS—in both data ingestion and query efficiency.

The remainder of the paper is structured as follows: Section II discusses related works and how the Incremental R-tree structure relates to other approaches in the literature. Section III provides an overview of R*-tree and STR-tree structures. Section IV details the core algorithms employed in the Incremental R-tree structure. Section V presents experimental results evaluating the performance of the Incremental R-tree structure. Finally, Section VI summarizes the key findings of the study and discusses future research directions.

II. RELATED WORK

This section reviews various approaches to the problem of bulk operations on disk-based spatial index structures. A last subsection discusses techniques employed in different contexts, including main-memory and cluster-based systems.

A. BULK OPERATIONS WITHOUT ADDRESSING WORKLOAD ADAPTABILITY

R-trees, a data structure that has been the subject of extensive research, are bulk-loaded using either bottom-up or top-down strategies. Bottom-up approaches, favored in many database and GIS systems due to their simplicity and satisfactory query performance, are commonly used. For example, the STR-tree sorts a 2D spatial dataset first along the x-axis, then the y-axis, packing the data into leaf nodes. This process recursively continues up to the root [2]. Conversely, top-down methods utilize more computationally expensive algorithms to generate more efficient trees. For instance, the Top-down Greedy Splitting (TGS) method constructs the R-tree top-down, employing cost-optimized binary splits that minimize bounding box area [3]. Each split iteratively examines multiple sorting orders to identify the minimum-area partition. Subsequent research has yielded numerous refinements to STR-trees and TGS-trees [6], [7], as well as alternative approaches [4], [5], [8], [9].

Another line of research focuses on merging newly incoming data with the existing tree after some restructuring [18], [19], [20], [21], [22]. In [18], one of the foundational studies on bulk operations, a simple lazy buffering mechanism is introduced at all R-tree nodes across every level to accumulate incoming data before reorganizing the tree. The study proposes a layered approach for incrementally reorganizing the tree structure. Instead of immediately searching for a leaf node to insert new rectangles, the method collects a batch of rectangles and stores them in the root's buffer. When the buffer reaches capacity, a "buffer emptying process" is initiated, routing each rectangle down to the next buffer node. This process prevents cascading effects and ensures that no buffer overflows occur, as buffers are managed to avoid exceeding their limits. Overall, this lazy insertion strategy enhances efficiency in handling bulk insertions into the R-tree. Additionally, the study provides a rigorous theoretical analysis of the proposed algorithms and a comprehensive set of experiments demonstrating that the buffered approach results in efficient indexes following batches of updates in terms of query execution.

In [19], after constructing a small R-tree from the bulk data, it is merged with the existing R-tree using a bottom-up approach, maintaining spatial locality and balancing the tree structure. The method is optimized for localized bulk insertions but may not perform as effectively for global datasets where new data is distributed across the entire spatial domain.

The seeded clustering algorithm in [20] groups new data points into clusters, ensuring that spatially close data is packed into the same nodes during insertion. While clustering minimizes interference with the existing tree during bulk inserts, its effectiveness relies heavily on the quality of the clustering process. Poorly formed clusters can result in a suboptimal tree structure and reduced query performance.

The study in [21] presents a new merge-based algorithm for bulk insertion of point data sets into xBR+-trees which is a Quadtree-based balanced structure. Since the xBR+-tree is optimized for point data and uses Quadtree-inspired partitioning, the proposed method does not generalize well to datasets with more complex geometries, such as polygons or lines. More recent study in [22], previous bulk-insertion algorithms for xBR+-tree have been modified to achieve higher performance by taking advantage of the special features of Solid State Drives (SSDs).

In summary, this category of research aims for fast loading of static data and minimal disruption from bulk insertions. However, these studies do not directly address the influence of workload characteristics such as data or query skewness.

B. BULK OPERATIONS WHILE ADDRESSING WORKLOAD ADAPTABILITY

More recent studies focus specifically on the workload-aware construction of the R-tree, with some involving the initial learning of workload characteristics before building the tree accordingly [12], [13], [14], [15]. However, they do not address bulk insertions into the existing tree.

PLATON in [12] offers a state-of-the-art solution based on Monte Carlo Tree Search (MCTS) for top-down R-tree packing by introducing a learned, workload-aware partition policy. This approach explicitly tailors the R-tree structure to the specific data distribution and query workload, necessitating knowledge of workload patterns. To mitigate the exploration cost of MCTS and achieve linear-time complexity, two optimization techniques—early termination and level-wise sampling—have been implemented. Extensive experiments on both synthetic and real-world datasets using libspatialindex and PostgreSQL demonstrate that PLATON outperforms existing R-tree variants and other workload-aware indexes, such as LISA [13] and RLR-tree [14], under various workload scenarios, including various query aspect ratios and query sizes.

All these recent studies in [12], [13], [14], and [15] apply machine learning (ML) models in various forms for building R-trees, moving away from the hand-crafted heuristics used by traditional R-trees and their variants. While machine learning-based approaches seem promising, there appears to be a need for additional time before they can be effectively integrated into systems due to implementation challenges. Specifically, since learned indexes require training, their effectiveness with highly dynamic data remains uncertain. Methods such as PLATON and LISA are primarily designed for read-intensive applications and assume that data loading is performed offline. For example, it has been reported that PLATON requires approximately one hour to learn optimal splitting points at each tree level for a dataset of size 1e6, based solely on a given query workload. Similarly, the learned index structure LISA exhibits comparable offline training and loading times. As such, these approaches are not suitable for dynamic scenarios like those addressed in this study, where the objective is to incrementally ingest new data batches and adapt to evolving query regions in real time.

Waffle [16], an external-sort-based bulk loading method, ensures optimal disk page utilization for static data while emphasizing square-like nodes. Its algorithms for dynamic updates focus on individual inserts and deletes rather than bulk operations, utilizing a maintenance mechanism that monitors query workload and distribution to continuously optimize the index. Waffle defers index optimizations until they are necessary based on the query workload. Specifically, nodes frequently accessed by queries are reorganized more often than those subject to more updates. Consequently, different sections of the tree are maintained according to the workload associated with their corresponding areas in the data space.

Another study by the same authors of [16], FMBI (Fast Multidimensional Bulk-loaded Index) and AMBI (Adaptive Multidimensional Bulk-loaded Index) in [17], relies on scanning rather than external sorting of multidimensional point datasets. AMBI is presented as highly advantageous compared to the non-adaptive FMBI, particularly when queries cover a small part of the data space. In such cases, only a partial index is generated, which gradually converges to a full index. Similar to Waffle, new data is inserted individually but with a lazier strategy. When a new point causes a leaf node to overflow, a new page is allocated for that point and future insertions. The node is processed and refined only when queried, and underflows due to deletions are similarly addressed upon access. Unlike Waffle, this lazy approach does not require maintaining statistical information for each node, such as query and update counters. The study also presents parallel versions of FMBI and AMBI, suitable for distributed systems and spatial partitioning.

The Incremental R-tree in this study shares similarities with Waffle [16] and AMBI [17]. However, while the Incremental R-tree emphasizes distributing continuous bulk inserts to the tree leaves and integrating certain partitions in a query-sensitive manner, Waffle and AMBI optimize bulk loading and individual insertions based on query distribution. Additionally, their indexing framework is primarily based on space partitioning, in contrast to the data partitioning approach used in the R-trees.

C. OTHER RELATED STUDIES

Although they are main memory-based structures, QUASII [29] and AIR-tree [30] are notable contributions to the field of adaptive indexing. Both incrementally reorganize partitions based on the data and the queries, employing cracking operations in response to workload queries. The Query-Aware Spatial Incremental Index (QUASII) in [29] features a three-level nested tree structure that incrementally and locally reorganizes data based on the boundaries of a window query across each dimension. In this context, it operates similarly to database cracking along the x, y, and z dimensions. Database cracking, a foundational concept in adaptive indexing for columnar main memory databases, focuses on the incremental organization of sortable datasets based on range query bounds [31]. Additionally, the QUASII strategy is inspired by the STR-tree, as it organizes data along each dimension.

Similarly, it has been reported that fully organizing a neuroscience dataset for modeling the human brain into an R-tree is inefficient for query processing due to the high data volume and numerous overlaps [27]. Instead, it is suggested that objects in the dataset be roughly partitioned based on spatial proximity, followed by the construction of a seed R-tree on

these partitions. This allows for more efficient spatial range query processing through a two-phase approach: initially randomly accessing nodes of the seed R-tree, then performing sequential access on the subsets. This method has proven effective for both surface-level and deep exploration [27].

The concept of partially organized seed structures has been effectively adapted to cluster-based systems, as demonstrated by the seed k-d tree approach [11] for workload-aware incremental partitioning of large-scale spatial data. This method begins by organizing initial data into balanced subsets via the seed k-d tree, then incrementally incorporates new data through MapReduce while maintaining the existing structure. A query-sensitive cost model dynamically triggers targeted expansions only when subset imbalances are detected, avoiding full reorganizations. This approach elegantly reconciles the efficiency of batched updates with the adaptability required for streaming spatial workloads.

The challenge of continuously ingesting new data batches has also been widely addressed in large-scale distributed data systems through out-of-place update paradigms, most notably LSM-tree-based approaches [33], [34], [35], [36]. While in-place update strategies apply inserts or updates directly to their designated locations within the index, they often suffer from high random I/O costs due to frequent node splits, merges, or rebalancing operations—especially when handling large or skewed datasets.

In contrast, out-of-place update strategies temporarily buffer incoming updates (e.g., in logs, deltas, or staging layers) and defer their incorporation into the main index structure through periodic reorganization or merge operations. This strategy is particularly effective for write-heavy workloads, as it enables batching and reduces write amplification, although it may negatively impact read performance if updates are excessively delayed. Consequently, recent studies have proposed hybrid approaches that combine bulk-processing capabilities with incremental update mechanisms to balance throughput and query responsiveness [36].

Inspired by the batching and high-throughput capabilities of LSM-tree's out-of-place paradigm, this study adopts a similar philosophy. However, to meet the needs of real-time exploratory analysis on a single machine, the proposed method differs in that it accumulates new data batches temporarily, and then integrates them into the tree using in-place updates that are selectively applied—based on query relevance—to minimize random access overhead.

III. PRELIMINARIES: R*-TREE AND STR-TREE

Since the Incremental R-tree incorporates strategies from the STR-tree [2] and R*-tree [28] variants, this section will present the fundamental characteristics and comparisons of these structures.

The R-tree structure [1] is a hierarchical indexing method that facilitates efficient search operations in spatial datasets. It achieves this by progressively refining the search space from the root to the leaves, where each level represents a different level of granularity. At each level, data objects are grouped into clusters using minimum bounding rectangles (MBRs). An MBR is defined as the smallest axis-aligned rectangle that completely encloses either a data object (at the leaf level) or a set of lower-level MBRs (at internal nodes). This hierarchical organization allows for efficient pruning of the search space, as only those MBRs that potentially intersects the query region need to be examined.

For an efficient index in terms of query performance, the general principle is to achieve upward clustering resulting in nodes with small MBR areas, minimal overlap between nodes at the same level, and minimal dead space. In terms of storage space efficiency, the goal is to build the shallowest possible tree with fully packed nodes. Since the parameters involved are interdependent, numerous R-tree variants have been proposed, each employing a distinct set of heuristics and algorithms. Among the many variants of the R-tree, the R*-tree and STR-tree structures are widely recognized as the most effective. While R*-tree excels in indexing spatial datasets at 1-by-1 loading, STR-tree shines in bulk loading scenarios. R*-tree, despite its slower loading process, delivers superior query performance, whereas STR-tree, with its significantly faster loading time, still provides comparable query efficiency.

R*-tree improves the query performance primarily through its advanced node splitting and forced re-insertion strategies. For instance, the R*-tree considers the MBR margin along with area and overlap criteria when determining axis-aligned splits, encouraging leaf nodes to produce more square-shaped MBRs. This reduces overlap at higher tree levels and positively impacts query efficiency. Otherwise, R*-tree's forced re-insertion strategy addresses the non-deterministic nature of the R-tree. Instead of generating different trees for the same dataset based on insertion order, R*-tree re-inserts some old entries, making the tree structure more deterministic [28]. This strategy is achieved by distributing a percentage of the entries (typically the 30% farthest from the MBR's center) from a full node to neighboring nodes at the same level instead of splitting. This approach replaces local split operations with a more holistic and robust approach, resulting in a considerable improvement in storage efficiency.

In contrast, STR-trees employ a straightforward algorithm, constructing a tree by sequentially sorting the existing data along each dimension, resulting in fully packed leaf nodes. Recursive application of this principle yields a highly storage-efficient tree with very fast construction time.

The Fig. 1 below shows the leaf MBRs of the generated R*-tree and STR-tree structures for a dataset of 100 data objects distributed over a portion of the unit area, highlighting the difference between the two trees. In this small dataset, the example R*-tree and STR-tree structures have a capacity of 8 data objects per node, for visualization purposes.

As shown in the Fig. 1, the R*-tree employs a more elaborate strategy to generate more MBRs, reducing overlap and dead space, while the STR-tree prioritizes storage efficiency by sacrificing some criteria like overlap and dead space.

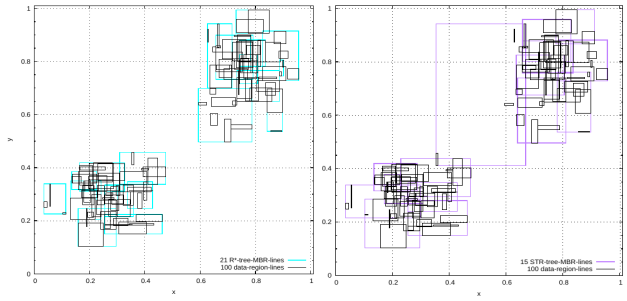


FIGURE 1. 100 data regions stored in 21 R*-tree leaves and 15 STR-tree leaves.

IV. INCREMENTAL R-TREE

In the Incremental R-tree, the existing dataset is initially bulk-loaded using a traditional STR-tree to form the seed structure. Subsequent incoming data batches are incorporated into this seed tree through a three-stage process:

Stage 1 – Grid-based Preprocessing: Incoming data batches are first partitioned into the buckets of a static fixed grid structure.

Stage 2 – Query-guided Overflow Placement: Partial results retrieved by window queries from the Grid File are distributed into existing leaf nodes of the tree as overflow buckets.

Stage 3 – Query-sensitive Overflow Integration: Some overflow buckets are then reassigned to newly generated leaf or internal nodes, guided by a cost model that is sensitive to query behavior.

This approach intentionally avoids immediate indexing of all incoming data. For example, data that arrives later may be indexed earlier if it is accessed more frequently, enabling workload-adaptive organization. The Incremental R-tree snapshot in Fig. 2 displays the key stages of the process. The following subsections describe the key components of the Incremental R-tree architecture: the data batch loading module, the cost tracking module, and the query processing module, which includes the overflow bucket integration mechanism.

A. DATA BATCH LOADING

Once the STR-tree is constructed as the seed, each subsequent data batch is incorporated using a strategy that permits ''overflow'' into the most suitable leaf nodes rather than executing traditional insertions. The fundamental principle of selecting the ideal leaf—namely, minimizing area enlargement in internal nodes and overlap in leaf nodes—is still respected. In addition, to support continuous querying of incoming data batches, the leaf MBR and necessary MBR updates along the path from the leaf to the root are performed. However, costly operations such as node splitting and reinsertion are deferred in favor of workload-aware processing.

Furthermore, individually inserting records into the tree should be avoided, as this naïve approach can result in excessive random disk accesses, significantly degrading I/O

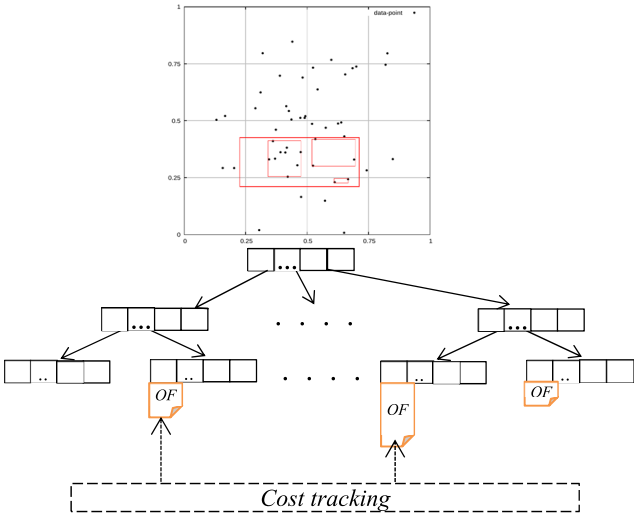


FIGURE 2. Snapshot of the Incremental R-tree: partitioning of a new data batch into grid buckets (Stage 1), batch-wise data distribution guided by query-driven partial MBRs (Stage 2), and cost-tracking–based control of overflow integration (Stage 3).

performance. Similarly, updating internal nodes to reflect the growing leaf MBRs after every single insertion would be prohibitively expensive.

Instead of record-wise distribution, it is more efficient to identify spatially coherent subgroups of points and allocate these collectively to the appropriate leaves. Since these subgroups may be implicitly guided by query activity, incoming data points are initially partitioned into the buckets of a static fixed grid. If corresponding buckets already exist from previous loads, the new points are appended to them. Subsequently, for each query window, the partial result sets identified within each grid cell are batched and inserted together into individual R-tree leaves. The MBR of each partial result set, derived from the respective grid cell, guides the selection of the target leaf node for batch insertion. For example, Fig. 2 illustrates the distribution of 50 newly arrived points into the buckets of a depth-2 static Grid File, and schematically depicts the MBRs of partial result sets collected from three buckets by a subsequent window query.

This strategy helps reduce both tree reorganization and random access overhead, potentially offering tunable control via the depth of the grid structure. Naturally, directly assigning these new data subgroups to the leaves may lead to a different MBR distribution compared to traditional record-wise insertions—possibly increasing overlap among leaf nodes. However, such degeneration is acceptable as long as queries do not target these regions. Once queries begin accessing them, deferred reorganization steps—such as more informed redistribution operations tailored to the current workload—can gradually refine and optimize the affected parts of the tree. As Fig. 2 (illustrating the Incremental R-tree's structure) shows, indexing new records necessitates their sequential passage through grid file buckets, then leaf buckets, before final integration into the tree.

TABLE 1. Notation.

Symbol	Description
qs	Queries accessing the leaf node in current workload
ds	Data size of the overflowing leaf, i.e. including data within the leaf and data in its overflow bucket
$\mathbf{res}_i$	Size of the result set generated by i.query in the qs
sel_i	Selectivity of the i.query in the qs set, i.e. the ratio of qualifying data items to the total number of data items in the leaf's overflow bucket
ρ	Area ratio of the leaf node MBR to its parent node MBR.

B. COST TRACKING

While the first query batch processing is ideal, it is clear that overflow buckets occurring in multiple leaves during subsequent phases will negatively impact query efficiency. To address this, the aim is to improve overall query efficiency by fully integrating leaves with overflow buckets—those that most negatively impact query speed—into the tree. Consequently, the cost tracking module is essential for keeping track of the cost metrics of the leaves along with the query workload trend.

Metrics for certain leaves accessed during each query execution are updated, particularly those with large overflow buckets (i.e., high OFthreshold), while leaves that are not accessed or have small overflow buckets are not tracked. More formally, by providing the metrics in Table 1, the overall cost of any leaf can be computed using the following equation (1):

$$\text{COST} = 0.5 \cdot \text{COST}_{\text{old}} + \rho \cdot \sum_{i}^{qs} i \cdot \text{res}_i \qquad (1)$$

The module identifies the leaf with the highest cost among those it maintains in a Priority Queue (PQ) after each query and removes it from the queue if it meets the following criterion: the leaf must have been accessed by at least one query from the current workload batch. Apparently, even though this maintenance operation occurs after each query, there may not necessarily be a selection of a leaf; in fact, this is often the case.

The reasoning behind the proposed model can be summarized as follows: First, the selectivity of a query within a leaf node, sel_i is defined as the ratio of the result set size, res_i generated from that leaf to the data size, $\text{sel}_i = \text{res}_i/\text{ds}$. This value provides an estimate of the query's intersection with the leaf's MBR. Incorporating this intersection size into the cost estimation allows for a more accurate assessment of the query's impact on this region. Since the size of the scanned overflow bucket directly affects the cost, the value $\text{sel}_i \cdot \text{ds} = \text{res}_i$ encapsulates two key cost factors. Furthermore, to better reflect the current workload and enable more accurate selection of the highest-cost leaf, increased weights are assigned to leaf accesses.

Secondly, the expansion of the leaf's MBR, along with the amount of data in the bucket, is also significant in terms of the leaf's overhead. Excessive expansion increases the overlap of this leaf with its sibling leaves, negatively impacting query performance. This is where the area ratio of the leaf node MBR to its parent node MBR, ρ comes in, providing insight into the amount of this overlap. Otherwise, finding the intersection amount with the MBRs of all sibling leaves for this information would be both costly and an unnecessary detail.

Thirdly, tracking workload change is achieved by assigning different weights to the leaf accessed in the current or previous query batches. That is, a leaf accessed in the current batch is twice as important as the previous one. Additionally, leaves that were frequently accessed in the previous query batch but not selected for integration, even though they might be selected for integration in the current batch, should be ignored until they are accessed again. This is also crucial for adapting to changes in workload.

Implementation of these strategies incorporates the following considerations: First, only leaf nodes with overflow buckets exceeding a specified OFthreshold are tracked; and leaf nodes selected for integration are removed from the (PQ). (Note that for leaves with small overflow buckets, a sequential search is acceptable, making costly overflow integration more conservative.) Second, the cost of a leaf that is accessed for the first time in a read-intensive phase and captured in the PQ (indicating that it was also accessed in the previous read phase) is updated by adding $\rho \cdot 1 \cdot \text{res}_i$ to half of the previous cost. Furthermore, a final scan of the priority queue (PQ) at the end of each query batch removes leaf node access history older than the previous batch, ensuring the R-tree's sensitivity is limited to the most recent two batches. These design strategies allow the aforementioned key metrics to be evaluated using a lightweight cost tracking module.

C. QUERY PROCESSING

The processing of any query involves two distinct search steps: one within the tree structure and the other within the Grid File buckets. First, the tree-based search identifies the leaf nodes intersecting the query MBR. This step involves accessing those leaves, updating their associated metrics, and, if integration is triggered, incorporating overflowed entries into the tree from the selected leaf. Following this, the Grid File search retrieves partial result MBRs from relevant buckets, which are then directly inserted into the tree's leaf nodes.

Through this mechanism, the full indexing cost of newly emerging data batches is effectively amortized over one or more future queries. Naturally, while these operations may increase the cost of individual queries—potentially to varying degrees depending on the number of result MBRs and the likelihood of overflow integration—they are expected to improve the performance of subsequent ones, thereby reducing cumulative query latency over time.

The overflow integration is the most performance-critical process, as it directly impacts query latency in the tree with extensive splits across multiple levels. To perform the integration, the following steps are taken: First, the leaf that

has been decided for integration needs to be accessed (as it may not be a leaf accessed by the current query). Then, the data records in the relevant bucket, along with the data records already existing in the leaf, need to be distributed to multiple new leaf pages based on their spatial proximity as efficiently as possible. For this purpose, the integration process begins by generating Level 0 nodes through sequential external sorting - first on X-axis coordinates, then on Y-axis coordinates of the dataset, consistent with STR-tree's level construction approach. The process then propagates the generated leaf nodes' metadata (including MBRs and node IDs) to their parent index nodes at Level 1. However, since the parent node may not be able to accommodate the newly born, potentially numerous leaf nodes, it may undergo multiple splits. If a single split is sufficient, it is performed using the R*-tree split algorithm. When multiple splits are needed, a predetermined fraction of the entries closest to the centroid of the overflowing index node's MBR are initially packed into new nodes using the STR-tree build-level algorithm. Secondly, the remaining entries (i.e. furthest from the center) are distributed to other internal nodes at the same level (Level 1) as much as possible. This distribution is no different from the forced reinsert operation of the R*-tree, which can make the new structure more inclined towards R*-tree behavior due to R*-style splits. Consequently, the R*-style node generating rate (say, distribution ratio (DR)) balances the overflow integration efficiency and building a more efficient tree having R*-style nodes.

The strategy mentioned for Level-1 is applied all the way up to the root of the tree. If a single split is not sufficient to handle overflow at the root, a completely new STR-tree (i.e. with all its levels) is constructed. This means the overflowing root is replaced by a new STR-tree containing the new root. Nevertheless, the scenario in which the tree grows by multiple levels simultaneously is rare. Typically, overflowing nodes are distributed throughout the tree in a balanced manner before reaching the root.

In summary, the overflow integration process benefits from the STR-tree's rapid spatial indexing and the R*-tree's refined split/redistribution logic, ensuring efficient execution.

V. EXPERIMENTAL RESULTS

The Incremental R-tree was implemented using the libspatialindex C++ library, integrating R*-tree and STR-tree algorithms interchangeably. The library's buffer and disk-based storage managers ensure persistence of large-scale datasets, and its extensible framework facilitated the addition of new structures. Grid File buckets and overflow buckets at leaves are managed via the library's TemporaryFile utility. The integration modules were embedded in libspatialindex, while the cost-tracking module operates separately through the IVisitor interface.

In the experimental setup, oscillating workloads are evaluated over both synthetic and real-world spatial datasets using different indexing structures. In the synthetic data experiments, data batches db_i and spatial window query batches qb_i are generated sequentially (db_0, qb_0, db_1, qb_1, etc.), contributing to dynamic workload behavior. Each data batch consists of spatial points with specific positional distributions (e.g., uniform or clustered) confined within a unit area or its sub-regions. Similarly, each query batch comprises spatial window queries of varying sizes, also distributed with certain spatial characteristics across the same domain.

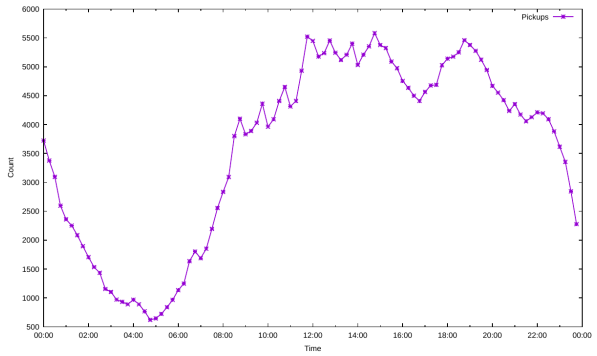


FIGURE 3. Hourly variations in taxi pickup density throughout New York City during a 24-hour period on February 15, 2016.

The real-world dataset is constructed by bulk-inserting taxi pickup locations collected from NYC into the experimental system at regular time intervals [32]. Fig. 3 illustrates the temporal variation and intensity of taxi pickup activity in NYC throughout the day. In these experiments, spatial data collected every m minutes is normalized into the [0, 1) unit square and bulk-loaded into each index structure. Immediately following each insertion, a set of spatial window queries is executed. The queries can either be exploratory, targeting the entire city area, or focused, aiming specifically at regions of active taxi pickup behavior. Fig. 4 presents a map-based visualization of taxi pickup activity during the 15-minute interval from 05:00 to 05:15 on February 15, 2016, followed by 30 focused window queries targeting high-density pickup regions.

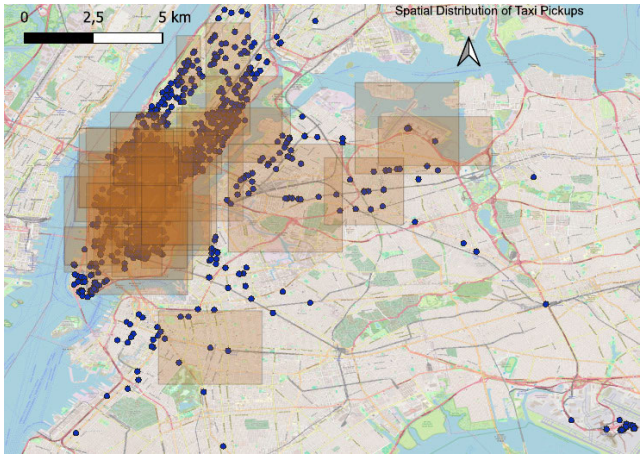


FIGURE 4. Map-based visualization of taxi pickups between 05:00 and 05:15 on 2016-02-15, along with 30 focused window queries.

To evaluate both the superiority of the Incremental R-tree over traditional spatial indexing structures and to assess

its adherence to the proposed cost model and overflow bucket integration strategy, comparisons were made against Table Scan (TS), R*-tree, STR-tree, STRof-tree, and the GiST-based R-tree (GiST-R-tree). The TS method processes the initial query workload directly on the first dataset and continues by appending each subsequent data batch to the same flat file. The R*-tree, once initialized through dynamic insertions, continues its growth while maintaining its established heuristics. The STR-tree begins with a bulk load and transitions into an R*-tree structure thereafter. The STRof-tree, in contrast, preserves growing external buckets without integrating newly arriving data into the internal tree structure.

Finally, the GiST-R-tree, built upon PostgreSQL's extensible GiST framework, enables efficient spatial indexing via MBRs and delivers high-performance query processing—making it a core component of PostGIS's spatial infrastructure. To optimize performance under highly dynamic and oscillating workloads, several best practices were implemented. All database operations were conducted using a single persistent connection to minimize connection overhead. The initial dataset was bulk-loaded into a PostGIS-enabled PostgreSQL table via the ''COPY mytable FROM STDIN WITH CSV'' command. A GiST index was then built using ''CREATE INDEX mytable_geom_idx ON mytable USING GIST (geom)''. To reduce query overhead during batch evaluations, window queries were executed as prepared statements: ''SELECT id FROM mytable WHERE ST_Intersects(geom, ST_MakeEnvelope($1, $2, $3, $4))''. Alternative index maintenance strategies, such as ''CLUSTER'', were considered; however, dropping and rebuilding the index after each subsequent ''COPY'' operation was ultimately chosen as it provided superior performance in this highly dynamic environment despite its potential cost. Following both initial and subsequent index reconstructions, ''VACUUM ANALYZE mytable'' was executed to update query planner statistics.

The performance of the structures under different exploratory scenarios (i.e., varying data batch and query batch positions) was evaluated. Evaluation metrics included the total time spent during a read-intensive phase, the total time spent during a write-intensive phase, and cumulative latency, which indicates the accumulated query response time. Latency was selected as the primary performance metric instead of I/O count because it more accurately reflects the performance implications of the index. This choice is due to the library's external sorting routines utilizing dynamically allocated memory pages and a separate buffer pool, rather than the buffer pool shared by the index nodes, and the Incremental R-tree's reliance on file system operations for overflow bucket management.

Table 2 lists key parameters and their values used in the experiments.

The page size for both the custom buffer and the storage manager in the experimental library was set to 8 KB, aligning with PostgreSQL's default page size. Each page could accommodate up to 180 (MBR, ID) entry pairs, setting the tree *node capacity* to 180. Other R*-tree parameters, including Fill Factor, Split Distribution Factor, and Reinsert Factor, were kept at their default values.

TABLE 2. Values for key properties.

PROPERTY	Value
Page Size	8 KB
Node Capacity	180
Buffer Pool Size	Enough to store almost all index nodes, excluding the leaf nodes.
FillFactor	0.7
SplitDistribution Factor	0.4
ReinsertFactor	0.3
Distribution ratio (DR)	0.5
OFthreshold	2
GridDepth	4

To improve experimental realism, a buffer pool large enough to accommodate nearly all internal (non-leaf) nodes was allocated across all evaluated tree structures. For instance, in the experiments presented in this study, the resulting trees comprise approximately 100–200 internal index nodes. Accordingly, the buffer pool size for both libspatialindex and PostgreSQL (via the shared_buffers setting) was configured to hold 150 pages, amounting to 1.2 MB (150 · 8 KB). The default random eviction policy of libspatialindex was used for buffer management, whereas PostgreSQL applies a clock-sweep algorithm—a lightweight approximation of LRU—for managing its shared buffer pool.

For external sorting, libspatialindex was configured with pages capable of holding 1,000 records each and a buffer pool of 100 such pages. Given that each index entry (containing an ID, 4 MBR corner coordinates) occupies 44 bytes, this setup results in a memory footprint of approximately 4 MB. These parameters were sufficient for single-pass external sorting during most initial load operations. For overflow bucket distribution, smaller values were preferred to conserve memory. To maintain consistency, PostgreSQL's maintenance_work_mem—which governs memory usage for operations like VACUUM and CREATE INDEX—was also set to 4 MB.

DR parameter, and OFthreshold in particular, are critical for fine-tuning the performance of the Incremental R-tree. The parameter OFthreshold, which defines the minimum overflow volume required for a leaf node to be considered trackable for potential integration, is closely tied to the available disk I/O bandwidth and is thus influenced by the underlying storage device and interface type. It serves as a critical control point for balancing indexing overhead and query performance in the Incremental R-tree.

Setting this threshold too low, particularly in environments with high disk read bandwidth (where scanning overflow buckets is relatively inexpensive), may cause the structure to prematurely enter integration mode, leading to excessive reorganization and increased cumulative query latency.

Conversely, configuring OFthreshold too high can significantly reduce the number of node splits, but at the cost of prolonged overflow effects that degrade query responsiveness, especially in high-access regions.

Based on empirical evaluations conducted on a typical SATA-interfaced SSD system, an OFthreshold value of 2 (i.e., twice the node capacity) was found to provide a satisfactory trade-off for a range of exploratory search scenarios. On the other hand, the *DR* parameter determines the balance between ingesting the overflowed nodes through the upper nodes and distributing them to sibling nodes during integration. This sometimes impact whether the tree grows more like an STR-tree or an R*-tree, consequently affecting cumulative query latency. In the experiments, this value was set to 0.5 to maintain a balance between performance and quality.

Finally, the optimum GridDepth value was determined through parameter sweeping. Reducing the grid depth results in fewer, larger buckets, which lowers the organizational overhead and accelerates the assignment of data to tree leaves. However, this also leads to greater structural disruption in the existing tree, negatively impacting query performance. Conversely, increasing the grid depth results in more, smaller buckets, which raises the cost of bucket organization and data distribution, but causes less disruption to the tree, preserving query performance. For instance, a depth of 4 (i.e., 256 buckets) yielded the best overall performance for all experiments shared in this study.

A. SAMPLE EXPLORATION SCENARIOS AND LEAF MBR PROGRESSIONS

This section analyzes the changes in leaf MBRs of trees in various exploration scenarios. To aid in visualization and validation, the batch sizes of both synthetic data and queries have been kept small. The first data batch contains 4,000 points, followed by two additional batches, each containing 2,000 points. Each batch is normally distributed within a specific quadrant. Query batches contain 20 uniformly-distributed $0.1 \cdot 0.1$ window queries. These scenarios represent some of the exploration cases that will also be used in the subsequent performance evaluation section.

Assume four main quadrants are designated as SW (0), SE (1), NW (2), and NE (3). The sub-quadrants can then be designated by concatenating these values. In the surface-level exploration scenario (see Fig. 5), the data batches follow a 0, 1, 22 pattern. Conversely, the query batches progress from 02 to 01 and finally to 22. Fig. 5 also illustrates the query results for the Incremental R-tree.

Fig. 6 presents the leaf MBR states observed at the end of each cycle for the R*-tree, STR-tree, STRof-tree, and Inc-R-tree, respectively. As illustrated, the R*-tree and STR-tree immediately index each newly arriving data batch, whereas the STRof-tree and Inc-R-tree selectively index only the portions of data retrieved by queries from the Grid File buckets.

For instance, in the second cycle, a data batch arriving in the first quadrant remains largely unindexed, as it is only sparsely accessed by queries located in quadrant 01. As a result, the corresponding leaf MBRs in that region expand only slightly.

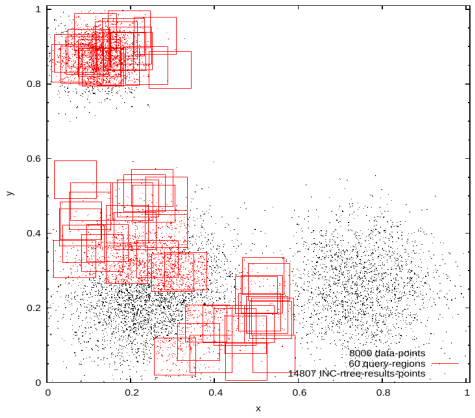


FIGURE 5. A surface-level exploration scenario.

Conversely, in the third cycle, the data batch arriving in quadrant 22 coincides with active query regions. Therefore, both the STRof-tree and Inc-R-tree index this region in response to the query load. Notably, while the STRof-tree does not address overflowed regions at all, the Inc-R-tree shows clear efforts to incrementally integrate the accumulated data piles into the tree structure. Thanks to the selective omission of regions that were not queried, the Inc-R-tree was able to maintain a much smaller index footprint—requiring only 66 leaf nodes, in contrast to the significantly larger indices maintained by the R-tree and STR-tree.

The in-depth exploration scenario (Fig. 7) follows a distinct spatial sequence: the first data batch covers the entire search area, followed by quadrant 3, and finally region 30. Query batches replicate this loading pattern. As shown in Fig. 8, the Inc-R-tree maintains a more conservative structure - ultimately producing 83 leaf nodes compared to the STR-tree's 114.

B. QUERY PERFORMANCE EVALUATION

This section presents the performance of the Incremental R-tree across various exploratory scenarios involving more data and query batches. Experiments utilized workloads featuring surface-level and in-depth exploratory characteristics, or their combinations.

Surface-Level Explorations: (Batches appearing sequentially in different quadrants at the surface level)

i. Query batches follows the same exploration pattern as the data batches.
ii. Query batches follows a different exploration pattern than the data batches.
iii. Data batches always appear in the unit area, while query batches are distributed randomly across quadrants.
iv. Both data and query batches always appear in the unit area.

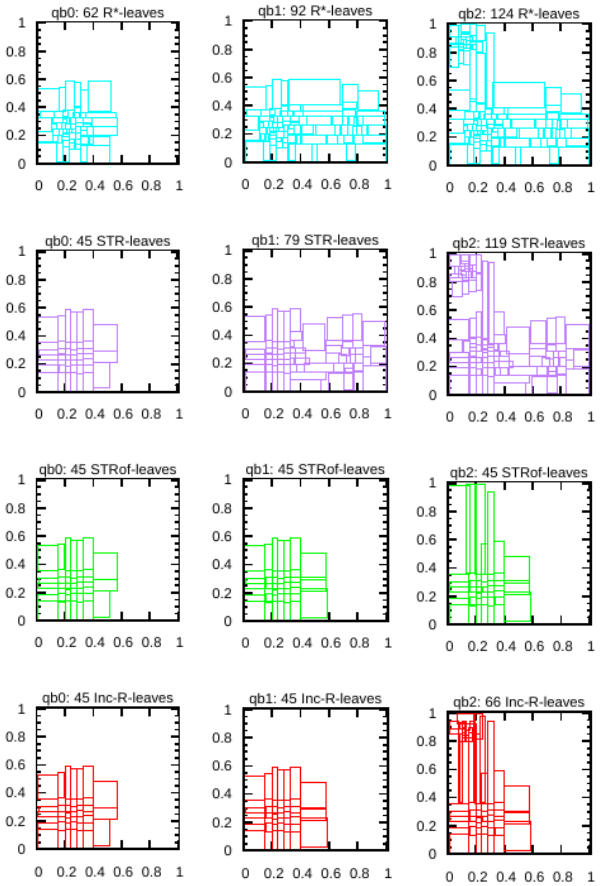


FIGURE 6. Leaf MBR changes of the R*-tree, STR-tree, STRof-tree, and Inc-R-tree under the workload, as shown in Fig. 5.

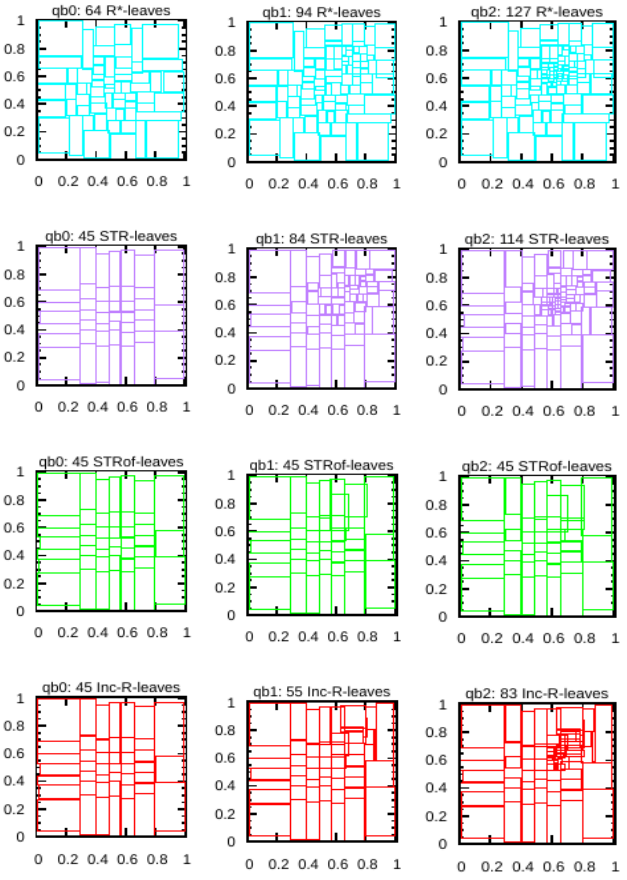


FIGURE 8. Leaf MBR changes of the R*-tree, STR-tree, and STRof-tree under the workload, as shown in Fig. 7.

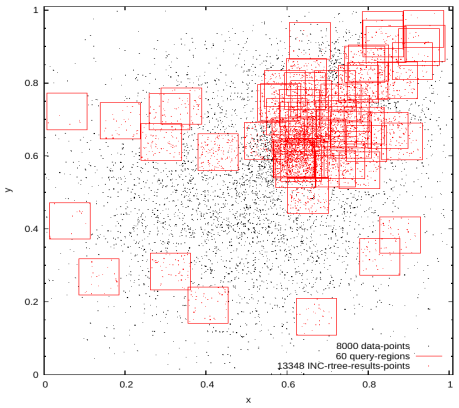


FIGURE 7. An in-dept exploration scenario.

In-Depth Explorations: (Batches appearing sequentially in progressively smaller sub-quadrants)

i. Query batches follows the same exploration pattern as the data batches.
ii. Query batches follows a different exploration pattern than the data batches.

The experiments were conducted using the system parameters summarized in Table 2 on a workstation equipped with an Intel® Core™ i7-7700 (3.6 GHz) processor, 8 GB RAM, and 64-bit Ubuntu Linux (4 KiB page size). For each experiment, the following metrics were measured:

- Total data loading latency per phase
- Total query latency per phase (to observe the immediate impact of data ingestion on subsequent queries)
- Cumulative query latency across all phases

Although the proposed method is designed to introduce only minimal delays during data loading, these delays may temporarily degrade the performance of the immediately following queries; however, the true benefit of the method becomes evident only when cumulative response time is measured over the entire workload. In all tested workloads, the Incremental R-tree demonstrated superior cumulative query latency compared to other methods, based on the common state of experiments conducted over multiple iterations. Only selected experimental results are presented in the paper.

The first two experiments are based on synthetically generated datasets. The first workload consists of oscillating patterns involving 5 data and query batches, while the second workload comprises 10 such phases. In each experiment, the initial dataset db_0 contains 10^6 points, while subsequent datasets (db_i, $i = 1$ to 4) each contain 10^5 points, generated with a normal spatial distribution within a quadrant

determined by the scenario. Each query workload qb_i consists of 500 queries. The queries, distributed according to a normal distribution within a quadrant specified by the experimental scenario, are represented as rectangles with a fixed area of 0.01 square units. The final experiment specifically investigates the behavior of the structures under more irregular (i.e., bursty) data conditions. Given the limited availability of suitable real-world datasets in the disaster management domain, a real-world dataset derived from NYC Yellow Taxi pickup locations has been adopted instead.

Workload #1: This workload is characterized by a *surface-level* exploratory nature, with query batches following data batches in the first four phases, while in the final phase, the query pattern departs from the data. Specifically, the data batches exhibit the pattern: 0, 3, 22, 1, 23, while the query batches follow a slightly altered pattern: 0, 3, 22, 1, 0. Table 3 and Fig. 9 shows the cumulative query response times, while Table 4, Table 5, and Fig. 10 illustrates the total latency for each method during the data-batch and query-batch phases. In the TS method, there is no latency due to data reorganization in the existing or newly incoming dataset; thus, only the delays caused by sequential search are observed. (See Fig. 10) Notably, it is only around the 300th query that the R*-tree, due to its excellent organization efforts, surpasses the TS in cumulative query latency, as seen in Fig. 9. Other methods achieve better results compared to TS (with perhaps the exception of the first 2-3 queries) due to lighter organizational loads during data loading and improved search performance through pruning in query processing.

It is important to note that the STRof-tree appears to slightly outperform the Inc-R-tree during the first query batch. This can be attributed to the fact that the STRof-tree incurs no overhead beyond collecting and organizing overflow buckets from the Grid File, as it performs no cost tracking. However, this early advantage highlights not only how lightweight the cost-tracking mechanism of the Incremental R-tree is, but also how critical it becomes in subsequent cycles. As the STRof-tree continues to accumulate increasingly bloated overflow buckets, it exhibits a steeper increase in cumulative query latency. Another key strength of the Incremental approach lies in its ability to suspend the bulk-loading cost of incoming data batches, which—unlike other methods—is not incurred as a sharp latency spike, but is instead distributed gradually across subsequent queries. This contrasts with the immediate performance penalties faced by traditional approaches during large scale data ingestion. This is where the significance of the Incremental R-tree becomes clear: it completes the experiment approximately 79% faster than the STR-tree, 49% faster than the STRof-tree, and 41% faster than the GiST-R-tree (see Table 3).

In the GiST-index evaluations, only the time required to construct the index was included in the latency measurements during the initial cycle, while the time spent loading data into the PostgreSQL table was excluded. Despite this, the initial indexing latency for the GiST-index was still approximately 20% slower compared to the index construction time within the custom experimental library. In the subsequent cycles, latency measurements encompassed both the data loading phase (via the COPY command) and the full index reconstruction process.

TABLE 3. Cumulative response times (sec) for Workload #1.

TS	GiST-R-tree	R*-tree	STR-tree	STRof-tree	Inc-R-tree
3014	54	424	152	63	32

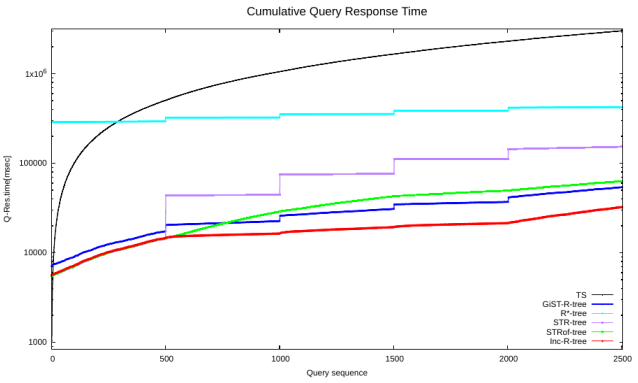


FIGURE 9. Cumulative latency for Workload #1.

TABLE 4. Total loading times (sec) per batch for Workload #1.

	TS	GiST-R-tree	R*-tree	STR-tree	STRof-tree	Inc-R-tree
db0	0	6.9	288.5	5.4	5.5	5.6
db1	0.4	3	29.9	29.3	0.1	0.1
db2	0.4	3.2	29.2	30	0.1	0.1
db3	0.4	3.7	30.6	34.4	0.1	0.1
db4	0.4	4.5	30.7	31.9	0.1	0.1

TABLE 5. Total query response times (sec) per batch for Workload #1.

	TS	GiST-R-tree	R*-tree	STR-tree	STRof-tree	Inc-R-tree
qb0	504	10	5.1	8.9	8.9	8.9
qb1	550	2.2	0.7	0.7	14.1	1.6
qb2	603	5	1.9	1.9	13.8	2.7
qb3	651	2.3	0.7	0.8	6.6	1.9
qb4	704	12.4	6.2	9.1	13.3	10.7

Fig. 10 provides a detailed breakdown of the performance improvements achieved by the Incremental R-tree. While the R*-tree and STR-tree achieve low per-batch query latency, this comes at the cost of significant delays during data loading phases. In contrast, both the STRof-tree and Incremental R-tree maintain minimal latency during data batch loading. However, the Incremental R-tree delivers superior overall query response times, attributable to its effective integration of overflow buckets.

Fig. 11 displays query response times after the 490th query. The plot reveals that each new data batch loading negatively impacts query latency. However, through its efficient

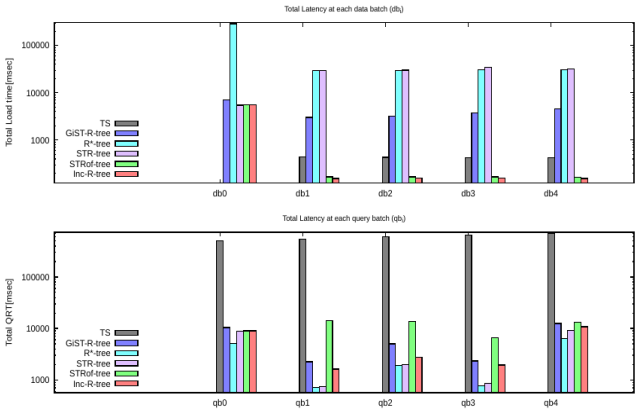


FIGURE 10. Total latency for each data batch (db_i) and query batch (qb_i) in Workload #1.

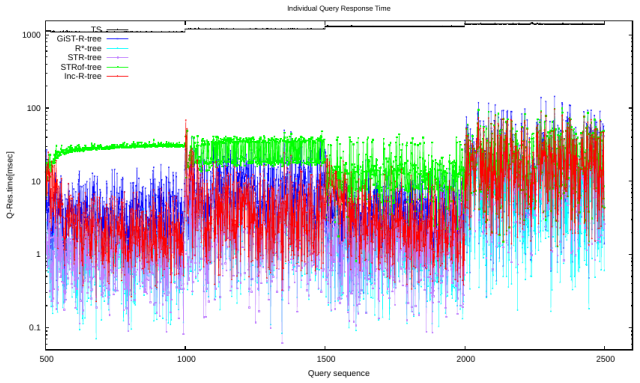


FIGURE 11. Individual query response times for Workload #1.

cost-triggered splitting mechanism, the Inc-R-tree's delays quickly converge to GiST-R-tree levels. The latency spike in the final cycle occurs when the query region moves away from the data insertion area, resulting in insufficient splits for tree reorganization - a pattern also observable in Fig. 9's final cycle.

Under this workload, the Inc-R-tree executed 253 splits following initial data loading, producing a final structure of 7,349 nodes. Including overflow pages, the total index size reached 60 MB. In contrast, the GiST-R-tree maintained a more compact 55 MB footprint. Both indices preserved a constant height of 3 throughout all experimental phases. Table 6 provides a comprehensive comparison of their storage characteristics.

TABLE 6. Index storage volume after Workload #1 execution.

	GiST-R-tree	R*-tree	STR-tree	STRof-tree	Inc-R-tree
NumOf Nodes	7092	10,827	9271	5651	7349
Size (MB)	55	89	76	46	60

Workload #2: This workload exhibits a hybrid exploratory pattern, combining surface-level analysis in the first phase

TABLE 7. Cumulative response times (sec) for Workload #2.

GiST-R-tree	R*-tree	STR-tree	STRof-tree	Inc-R-tree
133	607	344	290	61



FIGURE 12. Cumulative latency for Workload #2.

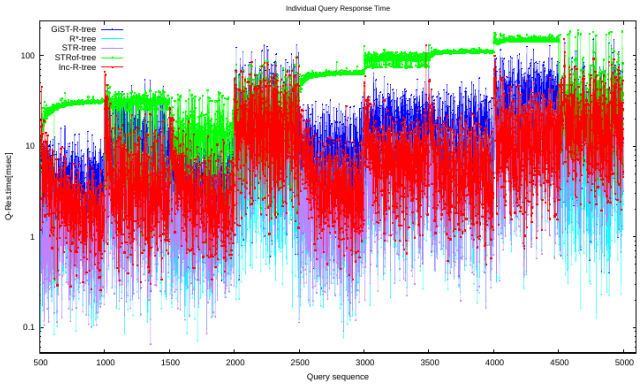


FIGURE 13. Individual query response times for Workload #2.

with in-depth examination in the final phase. Specifically, the data batches exhibit the pattern: 0, 3, 22, 1, **23**, 3, 30, 31, 311, 0 while the query batches follow a slightly altered pattern: 0, 3, 22, 1, **0**, 3, 30, 31, 311, 0. The purpose of this workload is to test the resilience of the Incremental R-tree in longer-duration workloads. Compared to Table 3, Table 7 shows that doubling the workload duration approximately doubles the Incremental R-tree's cumulative completion time. Fig. 12 and Fig. 13 show that the Incremental R-tree does not lose performance success in the later phases and maintains a steady performance. On the other hand, it is observed that the performance degradation of the STRof-tree increases more rapidly after the fifth phase. Notably, as observed in previous experiment, query latency shows moderate degradation when the exploration pattern diverges from the data loading pattern (e.g., during cycle 5 in this workload).

Under this workload, the height of all trees remained constant at 3 after the initial loading and did not change during subsequent data loads. Table 8 provides a comprehensive comparison of their storage characteristics.

TABLE 8. Index storage volume after Workload #2 execution.

	GiST-R-tree	R*-tree	STR-tree	STRof-tree	Inc-R-tree
NumOf Nodes	9659	14681	13746	5651	10265
Size (MB)	75	120	112	46	84

TABLE 9. Cumulative response times (sec) for Workload #3.

GiST-R-tree	R*-tree	STR-tree	STRof-tree	Inc-R-tree
114	130	129	121	61

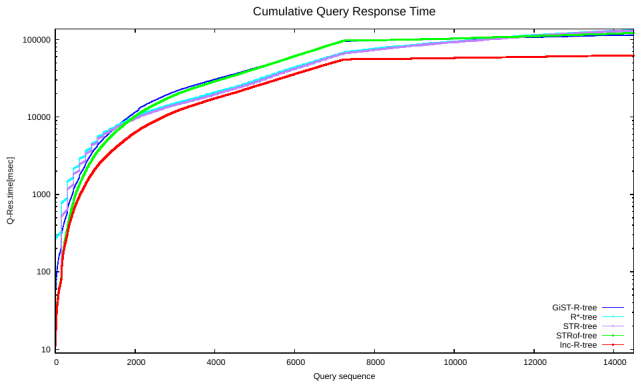


FIGURE 14. Cumulative latency for Workload #3.

Workload #3: The generated workload consists of loading taxi pickup locations collected at 15-minute intervals, followed by a batch of 150 spatial window queries. In the early morning hours, each data batch contains approximately 500 pickup locations per 15 minutes, while this number increases significantly to around 5,000 per 15 minutes during the afternoon and evening, up to 22:00. The purpose of this workload is to evaluate the resilience of the Incremental R-tree under bursty data patterns, where the volume of incoming data fluctuates significantly over time. The query batches are focus-based—i.e., window queries centered around actual pickup locations—until 12:00 PM, particularly targeting the steadily increasing traffic starting from 05:00 AM. After 12:00 PM, the query batches shift to a surface-based strategy, covering broader city regions with less localized targeting.

Unlike the previous experiments, this scenario evaluates a total of 96 cycles, each involving the ingestion of smaller data batches into the index structures. Additionally, in contrast to earlier setups, the GiST-R-tree was not dropped and rebuilt in each cycle. Since the amount of newly inserted data per cycle was relatively small, the index was left intact, and PostgreSQL's internal mechanisms automatically maintained it during COPY-based insertions. This resulted in improved performance for GiST-R-tree under this workload.

As shown in Table 9 and Fig. 14, the Incremental R-tree demonstrates significantly better cumulative latency during the second half of the day. This improvement stems from

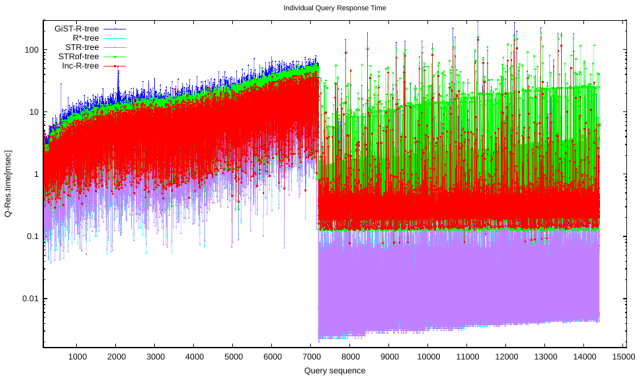


FIGURE 15. Individual query response times for Workload #3.

the nature of surface-based queries, where result sets tend to be smaller. Consequently, the Inc-R-tree retains a substantial portion of incoming data within Grid buckets, avoiding unnecessary transfers to tree leaves and reducing expensive operations such as node splits.

Finally, as illustrated in Fig. 15, the individual query response times of the Inc-R-tree—though not always as low as those of the R*-tree or STR-tree—were generally comparable to those of the GiST-R-tree. Overall, this last workload further reinforces the adaptability of the Incremental R-tree to varying data intensities and query behaviors.

VI. CONCLUSION AND FUTURE WORK

The rapid proliferation of location-based, mobile, and/or GPS-based data processing platforms has led to an explosive growth in both data and query sets. This yields a bottleneck for traditional data structures. Therefore, the ability of well-known self-organizing data structures to adapt to highly dynamic workloads—characterized by variations in data and query distributions and potentially query type—has been a critical factor in enhancing query performance and overall database system efficiency.

To address this problem, this study investigates enhancing query processing performance for spatial data analysis applications under oscillating workloads. Oscillating workloads are characterized by alternating read-heavy and write-heavy periods, and further complicated by the irregular distribution of both queried and newly inserted data within the potential data range. The proposed concept, implemented using simple yet effective heuristics within the complex libspatialindex library, demonstrates that the Incremental R-tree outperforms state-of-the-art R*-tree and STR-tree algorithms* in terms of cumulative query latency across various exploration scenarios. Moreover, it consistently delivers better performance than the widely used GiST-based R-tree implementation in PostGIS, further highlighting its practical effectiveness under dynamic and oscillating workloads.

Therefore, it can be concluded that the Incremental R-tree significantly improves the adaptability of real-time spatial data processing systems to evolving and workload-sensitive environments.

The study fundamentally leverages overflow buckets at a simple grid structure and at the leaf nodes of the R-tree before fully indexing newly arriving records. Future work will explore several strategies to further optimize performance. One possible direction is to replace the current static Grid File structure with a lightweight but more adaptive alternative, such as a kd-tree, which can better capture data distribution without requiring parameter sweeping. This would make the preprocessing stage more responsive to spatial variability in the data.

In the full-indexing phase, beyond cost tracking for query-sensitivity, overflow integration could be performed more gradually, rather than in a single step. For example, instead of merely performing file scans on the accessed buckets, a concurrent stage of external sorting could be introduced during query execution. This kind of adaptive, query-driven by-product may reduce future bucket integration latency—analogous to database cracking techniques [31]. Furthermore, instead of integrating the entire bucket at once, a more incremental integration model could be explored, where overflow contents are gradually distributed among appropriate tree nodes. Finally, a similar model could be extended to support k-nearest neighbor (k-NN) queries, another widely used spatial query type.

Since overflow-based approaches inherently promote sequential access patterns, this opens up possibilities for adapting the proposed Incremental R-tree to distributed or parallel systems. For instance, the parallel execution of both the distribution of large-scale incoming data batches into grid buckets and their subsequent allocation to leaf nodes could improve disk and buffer utilization significantly.

Moreover, with its combination of overflow-centric design and workload-aware integration, the Incremental R-tree may serve as a viable alternative or complementary component to recently proposed LSM-based spatial indexing models [35]—particularly in systems requiring both write-efficiency and query adaptability. This direction represents a promising avenue for future work, especially in the context of scaling the structure to high-throughput, real-time spatial data ingestion environments.

Lastly, instead of relying solely on a priority queue with static heuristics, future versions of the Incremental R-tree could incorporate machine learning-based policies for selecting which leaf nodes to integrate. For instance, a reinforcement learning agent could learn optimal integration sequences based on observed query workloads, while supervised models could predict the utility of integrating specific overflow buckets using features such as query density, recency, and spatial locality.

ACKNOWLEDGMENT

The author acknowledges the use of large language models, including ChatGPT (OpenAI), in 2024, Gemini (Google), in 2024, and DeepSeek (DeepSeek Inc.), in 2024, solely for language editing, grammar refinement, and minor proofreading. No AI tools were used to generate technical content, figures, and code in this article.

REFERENCES

[1] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *ACM SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, Jun. 1984, doi: 10.1145/971697.602266.

[2] S. T. Leutenegger, M. A. Lopez, and J. M. Edgington, "STR: A simple and efficient algorithm for R-tree packing," in *Proc. 13th Int. Conf. Data Eng.*, Feb. 1997, pp. 497–506.

[3] Y. J. García, M. A. López, and S. T. Leutenegger, "A greedy algorithm for bulk loading R-trees," in *Proc. 6th ACM Int. Symp. Adv. Geographic Inf. Syst.*, Nov. 1998, pp. 163–164.

[4] I. Kamel and C. Faloutsos, "On packing R-trees," in *Proc. 2nd Int. Conf. Inf. Knowl. Manage.-CIKM*, 1993, pp. 490–499, doi: 10.1145/170088.170403.

[5] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi, "The priority R-tree: A practically efficient and worst-case optimal R-tree," in *Proc. ACM SIGMOD Int. Conf. Manage. data*, Paris, France, Jun. 2004, pp. 347–358.

[6] H. Alborzi and H. Samet, "Execution time analysis of a top-down R-tree construction algorithm," *Inf. Process. Lett.*, vol. 101, no. 1, pp. 6–12, Jan. 2007.

[7] B. C. Giao and D. T. Anh, "Improving sort-tile-recusive algorithm for R-tree packing in indexing time series," in *Proc. IEEE RIVF Int. Conf. Comput. Commun. Technol.-Res., Innov., Vis. Future (RIVF)*, Jan. 2015, pp. 117–122.

[8] H. Tan, W. Luo, H. Mao, and L. M. Ni, "On packing very large R-trees," in *Proc. IEEE 13th Int. Conf. Mobile Data Manage.*, Bengaluru, India, Jul. 2012, pp. 99–104, doi: 10.1109/MDM.2012.40.

[9] J. Qi, Y. Tao, Y. Chang, and R. Zhang, "Packing R-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability," *ACM Trans. Database Syst.*, vol. 45, no. 3, pp. 1–47, Sep. 2020, doi: 10.1145/3397506.

[10] D. Achakeev, B. Seeger, and P. Widmayer, "Sort-based query-adaptive loading of R-trees," in *Proc. 21st ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2012, pp. 2080–2084, doi: 10.1145/2396761.2398577.

[11] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah, "AQWA: Adaptive query workload aware partitioning of big spatial data," *Proc. VLDB Endowment*, vol. 8, no. 13, pp. 2062–2073, Sep. 2015, doi: 10.14778/2831360.2831361.

[12] J. Yang and G. Cong, "PLATON: Top-down R-tree packing with learned partition policy," *Proc. ACM Manage. Data*, vol. 1, no. 4, pp. 1–26, Dec. 2023, doi: 10.1145/3626742.

[13] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "LISA: A learned index structure for spatial data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 2119–2133.

[14] T. Gu, K. Feng, G. Cong, C. Long, Z. Wang, and S. Wang, "A reinforcement learning based R-tree for spatial data indexing in dynamic environments," 2021, *arXiv:2103.04541*.

[15] J. Qi, G. Liu, C. S. Jensen, and L. Kulik, "Effectively learning spatial indices," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2341–2354, Aug. 2020.

[16] M. H. Moti, P. Simatis, and D. Papadias, "Waffle: A workload-aware and query-sensitive framework for disk-based spatial indexing," *Proc. VLDB Endowment*, vol. 16, no. 4, pp. 670–683, Dec. 2022.

[17] M. Hussain Moti and D. Papadias, "Fast and adaptive bulk loading of multidimensional points," 2024, *arXiv:2409.09447*.

[18] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter, "Efficient bulk operations on dynamic R-trees," *Algorithmica*, vol. 33, no. 1, pp. 104–128, May 2002.

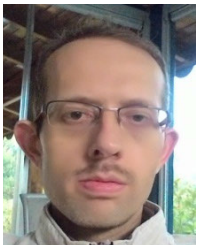
[19] L. Chen, R. Choubey, and E. A. Rundensteiner, "Merging R-trees: Efficient strategies for local bulk insertion," *GeoInformatica*, vol. 6, no. 1, pp. 7–34, Mar. 2002.

[20] T. Lee, B. Moon, and S. Lee, "Bulk insertion for R-trees by seeded clustering," *Data Knowl. Eng.*, vol. 59, no. 1, pp. 86–106, Oct. 2006.

[21] G. Roumelis, M. Vassilakopoulos, A. Corral, and Y. Manolopoulos, "Bulk insertions into xBR$^+$-trees," in *Proc. MEDI Conf.*, 2017, pp. 185–199.

[22] G. Roumelis, A. Fevgas, M. Vassilakopoulos, A. Corral, P. Bozanis, and Y. Manolopoulos, "Bulk-loading and bulk-insertion algorithms for xBR$^+$-trees in solid state drives," *Computing*, vol. 101, no. 10, pp. 1539–1563, Oct. 2019, doi: 10.1007/s00607-019-00709-4.

[23] S. Agrawal, E. Chu, and V. Narasayya, "Automatic physical design tuning: Workload as a sequence," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2006, pp. 683–694.
[24] L. Xing and W. G. Aref, "The AHA-tree: An adaptive index for HTAP workloads," 2024, *arXiv:2406.08746*.
[25] C. Zhang, G. Li, and T. Lv, "HyBench: A new benchmark for HTAP databases," *Proc. VLDB Endowment*, vol. 17, no. 5, pp. 939–951, Jan. 2024, doi: 10.14778/3641204.3641206.
[26] 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," *Proc. VLDB Endowment*, vol. 5, pp. 553–564, Aug. 2005.
[27] A. Stougiannis, F. Tauheed, T. Heinis, and A. Ailamaki, "Accelerating spatial range queries," in *Proc. 16th Int. Conf. Extending Database Technol.*, New York, NY, USA, Mar. 2013, pp. 713–716, doi: 10.1145/2452376.2452461.
[28] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," *ACM SIGMOD Rec.*, vol. 19, no. 2, pp. 322–331, May 1990, doi: 10.1145/93605.98741.
[29] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki, "QUASII: Query-aware spatial incremental index," in *Proc. 21st Int. Conf. Extending Database Technol. (EDBT)*, Vienna, Austria, Mar. 2018, pp. 325–336.
[30] F. Zardbani, N. Mamoulis, S. Idreos, and P. Karras, "Adaptive indexing of objects with spatial extent," *Proc. VLDB Endowment*, vol. 16, no. 9, pp. 2248–2260, May 2023, doi: 10.14778/3598581.3598596.
[31] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. Kersten, "Database cracking," in *Proc. 10th Int. Workshop Data Manage. New Hardw.*, Jun. 2014, pp. 1–8.
[32] NYC OpenData. (2017). *2016 Yellow Taxi Trip Data*. [Online]. Available: https://data.cityofnewyork.us/Transportation/2016-Yellow-Taxi-Trip-Data/uacg-pexx/about_data
[33] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996, doi: 10.1007/s002360050048.
[34] C. Luo and M. J. Carey, "Efficient data ingestion and query processing for LSM-based storage systems," 2018, *arXiv:1808.08896*.
[35] J. Shin, L. Zhou, J. Wang, and W. G. Aref, "An update-intensive LSM-based R-tree index," *VLDB J.*, vol. 34, no. 1, pp. 1–24, Jan. 2025, doi: 10.1007/s00778-024-00876-7.
[36] C. Luo and M. J. Carey, "LSM-based storage techniques: A survey," *VLDB J.*, vol. 29, no. 1, pp. 393–418, Jan. 2020, doi: 10.1007/s00778-019-00555-y.



MUSTAFA UTKU KALAY (Member, IEEE) received the B.S. degree in electrical and electronics engineering from Istanbul Technical University, in 1997, the M.S. degree in computer science from the University of Southern California, Los Angeles, in 2000, and the Ph.D. degree in computer engineering from Yildiz Technical University, in 2009.

He is currently an Assistant Professor and a Lecturer. His main research interests include spatial and spatio-temporal data management and adaptive index organizations.