Check for updates

Received: 1 February 2021 | Revised: 14 July 2021 | Accepted: 12 August 2021

DOI: 10.1002/cpe.6596

SPECIAL ISSUE PAPER

WILEY

Novel approaches on bulk-loading of large scale spatial datasets

Mustafa Utku Kalay

Department of Computer Engineering, Yıldız Technical University, Istanbul, Turkey

Correspondence
Mustafa Utku Kalay, Department of Computer Engineering, Yıldız Technical University, Istanbul, Turkey.
Email: ukalay@yildiz.edu.tr

Summary

Index structures that are bulk-loaded adaptively based on the query workload have been the subject of many database research, primarily in conventional databases holding one-dimensional data. Similar problems are also valid for spatial databases, and novel approaches are needed in these systems. In this study, we first consider the adaptive indexing approaches in a broad sense and then specifically investigate some possible forms of adaptivity on bulk-loading of large spatial datasets while retaining some core mechanics of well-known spatial data structures. In our first design, we examined how the internal configuration of the R-tree spatial index could be better for various workload properties, such as the workload's average aspect ratio. Our experiments observed that Adaptive Sort-Tile-Recursive (STR)-tree performs better than the original STR-tree. Our second design, the main contribution of this paper, focused on incremental indexing, another form of adaptivity. We concentrate on the data-to-insight latency and loaded spatial data faster within this multi-stage process. We used the Partitioned B-tree loaded with Morton codes of spatial objects to achieve this improvement by compromising spatial proximity. Finally, the response time of the first query in our design has been better than using the ordinary STR-tree index by a factor of 34%.

KEYWORDS

adaptive bulk-loading, index performance, R-tree spatial index, space filling curves

1 | INTRODUCTION

In database systems, the physical design significantly impacts the system's overall efficiency, such as query execution throughput and the average query response time. Notably, a physically fine-tuned system is expected to efficiently access the searched records with appropriate indexes on the search keys in a query workload. Data access time to searched records via an index is naturally based on the type of database (i.e., one-dimensional or multidimensional), index's internal structure, and the type of the query (i.e., point, range, window, k-Nearest Neighbor, and aggregation) since those parameters exhibit different file access characteristics. Recent innovations on mobile and distributed computing and their accompanying technologies such as IoT and Big Data expose well-known data characteristics, named 3V: Volume, Variety, and Velocity. 3V, of course, has bred new challenges in system implementation because data access characteristics of these technologies can be either very different or change rapidly. World events that are impossible to predict, such as sports, elections, and natural disasters, incur equally unpredictable burst traffic on the database backend. As a realistic example, to manage over 500 million tweets per day, Twitter uses MySQL database as a building block, upon which many sensitive tunings of data access strategies are applied.[1] In such environments, efficient fine-tuning of the system can only be performed with more flexible data (and index) structures with an internal structure keeping up with the workload. Another challenge comes from the rapidly increasing

 © 2021 John Wiley & Sons, Ltd.

amount of data today, making a bottleneck in the index loading latency. The idea of constructing the index incrementally has been a motivation for many studies to overcome this severe bottleneck.

Based on these observations, we can say that workload-aware query execution and automatic index creation/removal, and adaptive index loading concepts are among the current research topics in the database literature. When we look at the background of this research area, we come across different adaptive approach concepts in terms of adaptation to the workload. First studies seem to focus on determining the suitable index set.[2,3] While the workload is ready in advance in the *offline approaches*; the *online methods* are based on adding or deleting the indexes according to the observations during the workload processing.[4] The second one has a more dynamic aspect; however, keeping up with the workload was observed as a crucial bottleneck due to the high cost of index construction. The index might be utterly useless in the worst case, as the workload characteristics may have changed drastically until the index loading process is complete. Finally, more holistic approaches in which the index's internal structure is adaptive have been common in the last decade.[5,6] In those studies named *incremental indexing*, after the partial loading of the index according to the current workload, the incremental growth is controlled by only the queried parts of the data. The idea is based on organizing data partially when it is accessed during standard query processing. Partial organization in incremental indexes rather than "all or nothing" basis in first-generation studies seems like a more cautious approach as a by-product of query processing. Thus, incremental approaches aim to reduce the negative impact of rapid changes in workloads and the burden of unnecessary organization of unsearched data. Incremental indexing can be leveraged for both unpredictable and little predictable query workloads as well. For example, "questioning recent influenza activity more than older ones" or "questioning extreme fluctuations of influenza trends more than their stable values" are common types of workloads that can benefit from this approach.

Incremental index methodology can also be used in Exploratory Data Analysis systems, which usually analyze the statistical information belonging to a specific spatial region by visual means.[7] Such query systems have some distinctive characteristics in which ad-hoc queries come randomly after the previous query result, either in the previous queried region again (for a more detailed analysis) or in another area in space. Thus they are different from traditional systems where the workload is ready beforehand and be processed concurrently. Today, exploratory analysis is encountered in many scientific studies, from health to the space sciences. Examples come from many disciplines, such as detailed earth motion tracking models and detailed visualization of cell protein production models.[8] The common problem in these disciplines is the need to access big data efficiently. It is reported that within the Human Brain Project scope, it could take hours to load the spatial indices to query some spatial properties of approximately 1e11 neurons.[9] Starting the exploration as soon as possible can only be possible with incremental index structures that minimize the loading time and reach the ideal index gradually.

In this article, adaptive index structures, as a dynamic factor in system efficiency, are discussed for spatial databases with different aspects. Based on the motivations mentioned beforehand, there are two distinct contributions in this article. The first is bulk-loading spatial data into a well-known spatial index variant, STR (Sort-Tile-Recursive)-tree, by considering spatial characteristics of the workload, precisely the average aspect ratio of the query regions. This method is a kind of classic *offline approach* but differs from it in that it modifies the index internals based on the ready workload's characteristics. In our design, we considered the aspect ratio, which is an essential parameter in R-tree's efficiency,[10] in terms of workload in addition to data objects. With this intuitive and straightforward approach, we gain a better query execution performance without losing other quality parameters of the tree, such as loading time and size on disk. This concrete contribution is encouraging but minimal because while the gain is evident for uniformly distributed point datasets, some distortions occur with different parameters. However, we discuss issues rising with other general conditions such as non-uniform locational distribution of the data and query set and other types of spatial datasets (i.e., data with non-zero extents). Consequently, we indicate the design approaches with some concrete heuristics that could be a more comprehensive study in the future.

The second contribution, which seems more critical, is about the first stage of *the incremental index approach* mentioned beforehand as the third group of studies in adaptive indexing. The crucial point in the first stage of the incremental index is to fast load the raw data into the structure and retrieve the first insight as soon as possible. Accordingly, in this second part of our paper, we concentrate on improving the data-to-insight delay over large-scale spatial datasets. To achieve this goal, we strive for a "quick" way of taking the "big picture" of the whole dataset as an alternative to fully equipped static spatial index structures such as STR-tree. While STR-tree[11] and similar trees like Top-Down Greedy Split (TGS)[12] perfectly bulk-load the dataset for subsequent queries, such organizations are based on an all-or-nothing basis. There is no mention of enhancing the response time of the first query on the large-scale datasets with these conventional structures. Alternatively, mapping-based indexing techniques like Space-Filling Curves (SFCs) promises not the best but comparable query execution performance with less complexity in loading and query execution.[13] Because, in SFC indexing techniques, *data* in multidimensional space are mapped into one-dimensional linear ordered codes and then stored in conventional B-tree. Intuitively, relatively simple mappings of objects based on a predefined hierarchical traversal pattern in space may be used to have an idea about the proximity of objects. After that, *queries* are mapped to the sequence of sub-ranges in the linear ordering and searched in the B-tree. Moreover, instead of storing the SFC codes of all objects in a fully loaded B-tree in one step, we used the "Partitioned B-tree" as an infrastructure for an incremental index. Partitioned B-tree provides an incremental mechanism by partitioning the datasets into several batches and then merging those batches based on the incoming query ranges. In other words, instead of building a complete index that takes a lot of time, an index that may gradually improve is built. Finally, we achieved a better data-to-insight delay compared to loading the whole dataset into STR-tree. Although this part of our study is not a complete incremental index, the concrete contribution on data-to-insight delay is vital because the gain is beneficial not only for

the overall efficiency of the entire process but also for a stand-alone application. To complete the design of the incremental index, we showed some research directions as well.

Our article tried to demonstrate the benefits of our contributions with an experimental setup whenever possible. For the first contribution, we only used the "libspatialindex" library.[14] For the second contribution, we evaluated the performance gain by using two additional libraries[15,16]: Berkeley DB (BDB) and "libmorton." Spatial index library, written in C++, is an up-to-date library used in many articles on this topic and includes many variations of R-tree (i.e., R*-tree, R+-tree, STR-tree as spatial index variants, and Time Parameterized R-tree, Multi-Version R-tree as spatial–temporal index). Generic software design patterns inside the library facilitate measuring and comparing the query efficiency of index structures with different index internal configurations (such as node capacity, utilization). Additionally, the indices can be loaded with various datasets and workloads characteristics (such as uniform/skewed distribution, range/point queries) and under different physical constraints (such as disk page size, buffer pool capacity). The buffer and disk manager layers that allow disk access to be made over a buffer pool are also within the scope of the library. The extensible software architecture of the library provides for the implementation of novel index structures as well.

Our second tool, BDB, is not a complete database system, including many utilities. Yet, it has an embedded family of database libraries that enable storing key-value databases and provide high-performance data access services to applications. The system has many building blocks that can be plugged in and used with minimal administration. Since BDB has an architecture providing simple function-call interfaces to store and retrieve information quickly, client–server or inter-process communication is unnecessary. Thus, high-performance structures (i.e., B-tree and hashing) and other data management components such as ACID, reliability, and replication supports can easily be integrated into the applications.

The last library we used in our experimental setup is "Libmorton." SFC codes are generated by mapping multidimensional points to one dimension by interlacing the bits of the binary representation of each dimension. Libmorton generates Z-curve (Morton) codes for the low number of dimensions with high performance. Since we used two-dimensional *data*sets in our experiments, we do not need to apply more complicated algorithms such as the look-up table method, which is usually used in spaces with arbitrary dimensions. Lastly, since the library computes the Morton codes only for "points" in the space, we implemented a Quad-tree technique for "(Morton code) sequences of sub-ranges" corresponding to *query* regions.

The paper is organized as follows. The second section briefly describes spatial indices, specifically the R-tree and its dynamic/static loading. The third section presents a literature review on adaptive indexing, that is, workload-aware loading and incremental loading. In the fourth section, after introducing our workload-aware loading strategy, we demonstrate the improvements in the query efficiency provided by the proposed method. In the last sub-section, we discuss the possible shortages of our design (i.e., efficiency loss problems) and propose some hand-crafted heuristics and other potential research directions. In the fifth section, we introduce our second proposal for better loading large-scale spatial datasets via a "Partitioned-B-tree" and demonstrate the improvements on the data-to-insight criterion. In the last part of Section 5, we discuss the deficiencies in our design with two remarkable suggestions for future work. Section 6 concludes the study.

2 | PRILIMINARY CONCEPTS

Today, the most widely used index structure in database systems is the B-tree.[17] B-tree, a tree-based recursive and hierarchical structure on the "search key" containing either a single attribute or multiple attributes, is organized based on the total ordering of keys at all levels (i.e., leaf and internal nodes). Thus, B-tree is expected to manage sortable data within a single dimension. Essentially, the B-tree is a data structure that maps "search key(s)" to "record(s)" at a logarithmic run-time. B-tree is an indispensable data structure for database systems since it provides high space efficiency and significant speed gains in different types of searches (both of equality and range). At the same time, it maintains tree balance by dynamically organizing records while adding/deleting/updating operations.[18]

Spatial data is the representation of a point or region within two or more dimensional spaces. While "points" do not have any shape, "regions" may have arbitrary shapes or regular geometries (lines, rectangles, i.e.,) in one or more dimensions. Traditional query types or operations such as point or range query and join operations can be defined on spatial data similarly. However, their implementations require new data structures and algorithms since spatial data naturally cannot be sortable. Spatial datasets can be organized with spatial data structures by considering the criteria such as the proximity/relevance of objects to each other and their clustering in space. Thus necessary inferences can be made for different spatial queries. The ability to make these inferences in a compact data structure with accessing the least amount of data is undoubtedly essential in query efficiency. As can be predicted, efficient storage and access of spatial data play a vital role for many disciplines such as Spatial Databases, Robot applications, Geographic Information Systems, Computer Vision applications, and many others.

The dataset is organized recursively and hierarchically with two different types of space decomposition in spatial data structures, namely *space-driven* and *data-driven*.[19,20] In *space-driven* models, once the entire space is dynamically and gradually partitioned into cells (or grids), objects are assigned to relevant grids based on their positions. Cells can be accessed through a *trie* structure or a B+tree. B+tree can only be used after cells have been sorted by SFC such as Z-curve or Hilbert-curve. Examples of data structures in this approach are Region-QuadTrie, Octrie, MX(*Matrix*)-QuadTrie, PR(*PointRegion*)-QuadTrie, k-d-tree, and GridFile.[21] IBM DB2 and Microsoft SQL Server use this type of decomposition.

On the other hand, in *data-driven* structures, objects are grouped based on their locations and clustering, so there is no decomposition of all space. R-tree and its variants are in this group of indices. Oracle, Informix, PostGIS, and MySQL currently use this type of indices. Although R-tree is not arguably the best data structure for spatial data, it has many advantages in its widespread use and being a disk-based structure like B-tree and yielding a performance mostly independent of geometry. Thus we chose it as the index of focus in this article. We will briefly describe other structures (like SFC and Quad-tree) that we use as auxiliary in our study when it is appropriate.

2.1 | General features of R-tree

The dataset is decomposed into nested hyper-cube regions in an R-tree index structure depending on object locations.[22] For example, for two-dimensional data, the dataset is partitioned as nested rectangles. R-tree is a balanced and hierarchical tree, like a B-tree, but this time, a rectangle at each level is arranged to minimally surround the rectangles that are spatially close to each other at the lower level. Thus, while the tree's root becomes the largest Minimum Bounding Rectangle (MBR) surrounding all objects in space, the rectangles at the tree's leaves are the MBR of the data objects that may have arbitrary geometries. For two-dimensional data, the MBR is usually represented by the left-bottom and right-top coordinate values. Thanks to this compact organization of datasets, the cost of searching in different query types often depends on the tree's height. For example, in a containment query (finding objects containing a given *Q* point), the search starts from the root. It continues down to the lower level node(s), containing the *Q* point for a more detailed search at each level, and finally, reaches leaf node(s). In the intersection, k-NN, join, and similar queries, the search travels from the root to the leaves based on the query's selection criteria.

At its physical implementation, R-tree *non-leaf nodes* store a list of index entries, each of which holds the pair of information (denoted as [*P*, *R*]) about the sub-tree at the lower level. Here, the key-value, *R* in the pair, is the MBR of the sub-tree pointed by its accompanying pointer, *P*. In fact, each *R* is the MBR of several MBRs at the lower level grouped within a node based on spatial proximity. R-tree *leaf nodes* store a list of data entries, each holding the pair of information (denoted as [*P*, *R*]), like the non-leaf nodes. This time, each *P* in the leaf nodes points to the actual spatial information of the data (or the data record containing this information), and each *R* is the MBR information belonging to the spatial object. The number of *[P,R]* pairs in a node can be at least *m* and at most *M*. Usually, $m \leq M/2$ is applied due to node splits. While *m* is related to the space efficiency of the tree, *M, node's capacity* is associated with the height of the tree. Therefore, we can organize *N* spatial objects within an R-tree that has a height of $\lceil \log_M N \rceil$ at least.

The two essential properties of the tree structure, the space efficiency, and height are primarily based on the "order of data insertions," "data distribution in space," "*m*" and "*M*" parameters. In addition to that, three main criteria determine the quality of the R-tree[10]:

- Amount of empty space within each group (*dead space*).
- The amount of spatial intersection between groups.
- The proximity of the group's MBR to square (*aspect ratio*).

Each criterion has a relative impact factor on the tree's quality and is expected to be minimal during insert and delete operations.

As a representative example, Figure 1 shows the decomposition of $N = 11$ line objects into nested MBRs and their organization in a three-level R-tree with parameters of $m = 2$, $M = 4$. While A, B, D, and E are MBRs of leaf nodes from left to right, C and F are the MBR of the first level tree nodes. In this small example, the grouping of objects could take many different forms under the constraint of $m = 2$ and $M = 4$ objects in each group. Considering 8K-capacity (or more) of disk pages in today's real systems, it is clear that for higher *M* values (i.e., about 300, 400, or higher), different groupings based on some of the criteria mentioned above will significantly affect the *tree's space efficiency*, *occupied disk space*, and *query processing efficiency*.

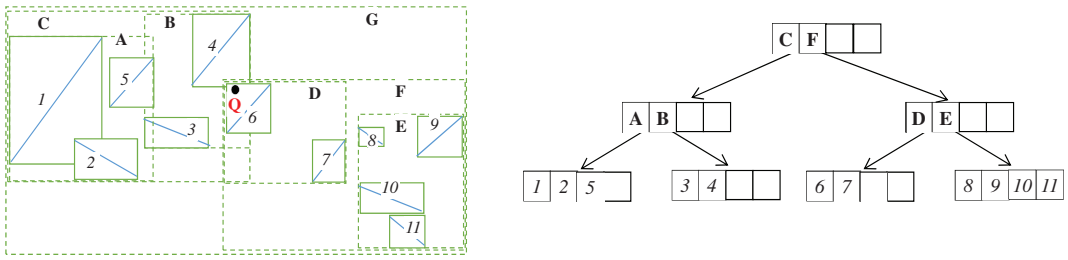


FIGURE 1 A sample grouping of 11 line-objects and the corresponding R-tree

In the subsequent R-tree variations, for example, all criteria together are evaluated in the R*tree.[10] Also, instead of splitting the node immediately when it is full, about 30% of the records in the full node are re-inserted into the tree to distribute the condensation to neighboring nodes. In another dynamic R-tree variation, R+tree,[23] it is preferred that some objects are clipped and inserted into two or more nodes so that the node MBRs of the same level are discrete. While this approach causes an increase in tree height due to the rise in the number of records, performance is positively affected in many query types. For example, since there is no MBR intersection in non-leaf nodes, the point query is assured to travel from the root to a single leaf.

We briefly reviewed the basics of R-tree implementation here. Still, the readers interested in more detailed information about the fundamental algorithms of dynamic R-trees may see the related papers.[10,19,20,22,24]

Instead of adding data into a tree one by one, named dynamic loading, to improve the insertion performance, there have been studies such as collecting data in a small R-tree and adding it to the big R-tree.[25–27] These methods, called "bulk insertion," can be considered a subclass of static bulk loading, which we will discuss next. Since real-life data often occur collectively, bulk-insertion yields more compact trees with much better loading times than one-by-one insertions.

With static data loading (namely "bulk loading"), all data are loaded at once. Now, it is possible to apply more robust heuristics and thus to build R-trees with much higher efficiency. For example, while the quadratic R-tree variation guarantees at least 50% space efficiency, this value in a typical R*-tree can only be 70%.[10,22] On the other hand, with bulk loading, it is usual to have much shallower trees with space efficiency close to 100%. The bulk loading process usually takes place based on sorting at each level. The tree may be built either from leaves to root,[11,28,29] or from root to leaves.[12] Hybrid approaches are available as well.[30]

In the first study on static R-tree,[29] object MBRs are sorted according to the values of their corners in only one dimension and written to the leaf nodes with 100% capacity. Afterward, the upper-level nodes are loaded similarly with the lower-level nodes' MBRs. This simple approach can be efficient for point queries by forming small and non-intersecting groups on the point dataset at the leaf level. However, since the group (node) MBRs have minimal aspect ratio values (i.e., far from square-proximity), the efficiency drops very quickly in range queries with high aspect ratios. Even worse, for random region objects, the amount of intersection at the leaf level increases uncontrollably. Hilbert-R-tree structure,[28] the first method that takes all dimensions into account, uses a SFC, Hilbert-curve, to partition the space into grid cells and sequentially store the data objects intersecting with grids into R-tree. SFC organizations provide a kind of traversing throughout the space with different patterns. (We also used the SFC technique as the baseline in our index loading design. See more elaborate definitions in Section 5) In the Hilbert R-tree bulk-load, the objects are loaded on the leaf nodes according to the Hilbert values of the selected reference points (e.g., center points or vertices). Once the leaf nodes are 100% filled, the loading continues until the root, where the MBRs of the previously built nodes are the *R*-entries of the parent nodes.

In bulk loading with STR-tree,[11] the tree is constructed from bottom to top, forming p^*p parcels (tree nodes) at each level. For example, for two-dim space, also valid for $d > 2$; the dataset containing N-objects can be stored in total $\sqrt{B} * \sqrt{B}$ number of leaf nodes with M-capacity, where B denotes the total number of nodes, $\lceil N/M \rceil$. So that each dimension has a common effect, space is organized such that there are $p = \sqrt{B}$ stripes, each having $p = \sqrt{B}$ nodes. To do this, first, the objects are sorted according to one dimension (e.g., x-axis) of their center points and divided into $\sqrt{B}$ stripes, each of which can contain $\sqrt{B}$ groups (nodes). Then, $M * \sqrt{B}$ objects in each stripe are sorted according to *the* y-axis this time and written into leaves as fully loaded from left to right. (Only the last page in the last stripe may be less than M objects) After that, the process continues recursively up to the root, with the upper-level nodes containing the node-MBRs loaded previously.

The algorithm complexity of both sort-based methods (Hilbert R-tree and STR-tree) is equivalent to external sorting in terms of Disk I/O criteria. If the main memory can hold b objects, with (b/M)-way merging the cost can be expressed as $O((N/M)\log_{b/M}(N/M))$. If we assume that N is large and $b \gg M$, the cost converges to $O((N/M)\log_b N)$, and it is much less than the complexity of one-by-one object insertion in a dynamic R-tree which has $O(N \log_M N)$ time-complexity. In addition, the fact that disk accesses in dynamic R-tree mostly show random characteristics, and accesses in static loading are usually in sequential style makes bulk loading much more attractive in terms of loading latency criterion.

In summary, in Hilbert R-tree and STR-tree, node MBRs close to the square are obtained, and thereby high query processing efficiency (although not the best) is achieved. Thanks to their easy implementation and easy integration into the systems, they are the most widely used (at least mentioned) packing algorithms in today's spatial databases (i.e., commercial or experimental) and geographic information systems. Consequently, these features of STR-tree and its current implementation in *libspatialindex* library have been the primary motivation for adopting this tree as the reference structure in our experiments.

As an alternative to bottom-up loading strategies, in TGS-based bulk loading,[12] input data are partitioned into subtrees in a top-down fashion. At each tree level, "all cuts" orthogonal to the coordinate axes that result in packed trees are considered. Additionally, trees are greedily picked based on optimizing an arbitrary "cost function." Although TGS-based bulk-loading always yields fine-tuned (near-perfect) structures, its high loading latency and implementation complexity has been mentioned in many studies.[24,28,31,32] In the first part of our study, we tried to inject the "query aspect ratio" parameter into bulk-loading, which seems more intuitive and easy for STR-tree. However, our strategy may also be possible for the TGS-based approach by optimizing the cost function according to the average query aspect ratio (AQAR). Although it has high loading latency, that would be an interesting study.

3 | LITERATURE REVIEW

This section reviews some innovative studies on adaptive indexing in the literature, specifically workload-aware bulk-loading and incremental indexing. In recent work, query-adaptive loading R-trees optimized in respect to a given query profile has been defined using a modified version of SFC rather than directly applying standard SFCs.[31] In the first part of their study, sorting the space (thus data objects) is performed within a "non-symmetric" pattern based on the workload profile for the cases where the average query shape is non-square. They apply different shuffling of bits for generating "non-symmetric" patterns. The second part of their study is about packing sorted objects into R-tree nodes and also quite remarkable. They argue that filling up pages to the maximum (a common approach) does not always lead to R-trees optimized concerning the given query profile. They propose a dynamic programming solution for query-adaptive partitioning of the sorted sequence of objects, rather than filling the nodes with full capacity from left to right in a traditional manner. They report an improvement in query performance at about five times for large data files, compared to conventional STR or Hilbert-order packing methods. Our first contribution has related to this work in terms of query-adaptive partitioning of the sorted sequence of objects. Unlike their approach based on SFC-based loading and dynamic programming, we adopted STR-tree-based approach. We tried to integrate the aspect ratio characteristic of the workload to refactor the groupings of data objects.

In contrast to static workload awareness, there have been trends for applying adaptations during the runtime of queries. The first remarkable study that we want to mention is a little older but reveals that the concept of adaptive index idea can be in many different forms. An adaptive index that improves overall query cost has been defined by allowing index nodes with various capacities (i.e., containing a variable number of disk pages).[33] The design is based on selectivity analysis of queries in the workload to reduce the random disk accesses. The tree's internal structure dynamically changes by allocating smaller capacities for nodes searched with point (or smaller range) queries, and larger capacities for nodes searched with larger window queries. They define a general framework applied to different types of trees (like B-tree and R-tree). While we used fixed-size disk pages in both index designs in our paper in a traditional manner, we usually analyzed the burden of random disk access and devised our algorithms with this fact in mind. (For example, our heuristics at the end of Section 4 and query execution in Partitioned B-tree in Section 5.2.) The structure in their study is not incremental in common sense as the adaptations are performed incrementally only after the index has been fully built. In other words, they did not concern with the cost of raw data querying (i.e., data-to-insight latency).

Another study in this category named "Query Responsive Index Structures",[34] Becker et al. defined two distinct variants of R-tree: Matrix-R-tree and Query R-tree to integrate the statistics collected during query execution into R*-tree. In the Matrix-R*-tree structure, they stored a counter Matrix within each node to identify the pairs of nodes with the most frequently accessed in incoming queries. However, in the Query-R-tree, they bundled an ordinary R*-tree (data tree) with another R*-tree that records the previous queries (and the frequency of their processing) on the data tree. Thus, the main idea is to track the frequently accessed node in Matrix-R-tree, and frequently issued queries in Query-R-tree. When a pre-defined threshold value is exceeded in both approaches, the response time in subsequent queries is improved by rearranging the data tree with different principles (based on better splitting or redistributions). The central intuition in their strategy is to enhance the performance based on the determination of the hot spots (i.e., frequently accessed nodes by previous queries) of the R*-tree preloaded ordinarily. Unlike their study, our first design is based on bulk-loading of R-tree according to the workload's geometrical characteristics, precisely average aspect ratio.

Another motivation of the adaptive index approach has been designing an index that can begin serving faster and gradually adapted to a complete index, so-called incremental indexing. Not surprisingly, the idea of incremental indexing has first spawned more or less a research field in single-dimensional databases. Column-based databases have been the first to implement incremental index structures under "database cracking."[6] In this approach, the attribute values are incrementally sorted in place based on the context-aware style of quick-sort, the well-known sorting algorithm. Essentially, pivot values used in quicksort are chosen as the search key values of point queries or limit values of range queries in the workload. This idea has been the basis for the last generation of column-based databases such as C-store and MonetDB, providing a high query efficiency. The weaknesses of database cracking are discussed with experimental observations.[5] Accordingly, it has been argued that achieving the final sorted list with cracking is very dependent on the order of search keys. Also, it unnecessarily has a high convergence time (i.e., time to reach a fully sorted list) due to its low fan-out (i.e., two or three at most) as a predicate can only crack the list at most two points and generates three sub-lists. Low fan-out mainly causes too much movement (i.e., swaps between partitions) of individual data records during the incremental transformation. Perhaps more importantly, since the organization in cracking is based on quick-sort in memory, this approach will be much less efficient (maybe inefficient) for block-access-based systems like disk and flash memory than merge-sort-based methods.

The need for incremental organization in a block-access-based manner (in traditional row-based systems) has been met by "adaptive merging."[5] Traditional external sorting is a standard way of sorting the large dataset that cannot fit in the memory. In this well-known algorithm, once the dataset is split into many "run" files, all the run files are *k*-way merged into a sorted dataset in some iterations with logarithmic complexity. In the "adaptive merging" method defined by Graefe and Kuno,[5] instead of generating the sorted list all at once with standard *k*-way patterns, merge iterations are controlled by only the queried parts of the list. To do this, a special B-tree named "Partitioned B-tree"[35] is used. Partitioned B-tree is initially constructed over the raw dataset by adding an "artificial leading key" to all records. Each distinct artificial leading key is used to define a partition (or batch) in the B-tree. In its initial form, the tree may have as many partitions as the number of pages in the file, while in its final form, it will have only one partition sorted entirely by the "real" search key. This transformation is based on an "incremental merging" of leaf nodes by collecting and merging only the queried parts in leaf nodes and moving them into a new partition (i.e., node(s)) as sorted. In this approach, since the initial B-tree

is very fast to construct, the time to start serving (i.e., data-to-insight latency) is less than the time needed to build a complete B-tree. Although the initial benefit taken from the partitioned B-tree is less than the one taken from a fully loaded B-tree, thanks to incremental merging with high fan-in (typically 100–500), with upcoming queries, the tree will converge to the stable form within a short time. Thus, the temporary query efficiency drop in the partitioned tree is compensated by the fast data-to-insight and eventually better system throughput contribution with this simple idea.[5] This is a straightforward idea because we may use the traditional B-tree structure without any code additions at the system level. Readers who want to see examples of incremental organizations with "database cracking" and "partitioned B-tree" can refer to related papers.[5,6]

Besides database cracking and adaptive merging, recently, more holistic approaches exploiting the most beneficial aspects of each technique have been investigated.[36] Alternative to database cracking and adaptive merging approaches, some other methods skip preprocessing to minimize the cost of raw data querying. For example, NoDB[37] and RAW[38] are prototype query engines that adapt themselves to the input data formats and queries. The incremental approaches we mentioned motivated us to consider these methods on spatial data. As we pointed out in the introduction, many emerging technologies and exploratory-based scientific studies need rapid access to raw spatial data.[7–9] Now, we shall review some approaches in the literature to designing an incremental index for *spatial datasets*.

Within designing incremental indexes, new criteria have been defined to assess the index[7(p327)],39 feasibility. Pavlovic et al.[7] has evaluated data-to-insight delay, cumulative execution time (i.e., related to the extra burden on the query processing due to the incremental organization), and the convergence time (i.e., related to the performance of the typical zooming behavior in the exploratory querying environment). They compared data-to-insight delay to searching delay over the raw data and processing the first query after total loading of the index. In their recent work, they define three paradigm examples of database cracking, named "SFCracking," "Mosaic," and "QUASII," all of which are memory-based organizations on a three-dimensional neuroscience database. SFCracking methodology is based on cracking the sorted *Z*-values of the dataset with each incoming query. However, unlike its one-dimensional counterpart, each query incurs many cracking points, and related reorganizations (swaps) as each query window generates several ranges of *Z*-code values.

In their second design, Mosaic, they incrementally build an Octree by dividing the octant that overlaps with the incoming query equally into eight segments. Thus, data in densely queried areas are re-partitioned multiple times. Mosaic, a simple incremental reorganization, unfortunately, has the inherent drawback of *space-driven* partitioning methods, namely "replication problem in data assignment." The problem is that objects are assigned to all partitions that they overlap with. The problem grows exponentially as the partitioning increases in large data volumes.

In their final design, QUeryAware Spatial, Incremental Index (QUASII), they reorganize data in a *data-and-query-driven* manner. QUASII incrementally and locally reorganizes data based on the window query's two boundaries at each dimension without performing a complete sort. In that sense, it is the database cracking at each dimension (i.e., *x*, *y*, and *z*, respectively). In the other sense, the strategy is inspired by STR-tree as the organization is performed on each dimension, *x*, *y*, and *z*, respectively. The significant difference is that the partitions are accessed via a three-level nested tree structure, each of which represents the cracking points along the dimension. Thus leaf nodes hold the most fine-grained partitions that are constrained by all dimensions. They described a depth-first query processing methodology in which data swaps (restructurings) are performed as a by-product. Their comprehensive experiments on neuroscience datasets modeled the workload as queries arriving sequentially in a single stream to simulate the exploratory analysis environment. Specifically, they generate five query clusters, each with 100 queries, where query centers are distributed around the cluster centers and run the workloads on different incremental organizations. They showed that QUASII might yield better response times for the first queries of the workload and a long-term profit on the query execution throughput. Unlike all these three designs, our second design in this paper is disk-based. Consequently, we adapted (Partitioned) B-tree instead of main memory structures like database cracking and Octree. The common point is that we focused on the efficiency of the first query (i.e., data-to-insight latency), which is their primary criterion in their evaluations.

The last study we want to point out is Space Odyssey,[40] a disk-based exploratory analysis system. It has been presented as an incremental structure enabling efficient querying on multiple spatial datasets of massive size. They focused in their study on a significant problem facing neuroscientists, that is, efficient access to some spatial ranges on subsets of multiple spatial raw datasets. In their study, neither what combinations of datasets nor what areas are queried in them are known beforehand. The second challenge is relevant to our study. They use incoming queries to reorganize the physical layout of the raw data via *space-driven* partitioning to serve the subsequent queries better. This is the disk-based version of Mosaic mentioned in the previous paragraph. The major drawback, the "data replication problem," seems to be a more severe threat to efficiency in the disk-based data processing.

In common sense, unlike all incremental-based studies mention above, in our second design in this paper, we concerned ourselves with only the data-to insight latency criterion instead of a complete incremental design. Yet, we suggested some essential research directions for a comprehensive incremental index.

4 | OUR WORKLOAD-AWARE LOADING STRATEGY FOR STATIC R-TREES

This section introduces our idea of making the R-tree loading more adaptable in an offline manner. Our approach applies to STR-tree with minor modifications of loading parameters based on the average aspect ratio of the ready workload.

4.1 | Data loading

Before starting our suggestion, some experiments showing the effect of static loading are presented. First, we compared dynamic and static loading of synthetic datasets using the spatial index library (libspatialindex). Datasets contain points that are uniformly distributed in the unit area of two-dimensional space. 4 KB disk pages and 10-page buffer pool were used in the experiments. For both R-trees, the capacity was chosen as $M = 92$. This value is the threshold value that fills the disk page. The occupancy ratio, a parameter to control space efficiency, was set to 0.99 for both trees. Although it is impossible to approach this occupancy level for the R*-tree due to arbitrary node splits, we tried to obtain the highest possible space efficiency for both trees. Figure 2 shows R*-tree and STR-tree loading delays depending on the dataset sizes. Instead of the number of disk I/O as the performance criterion, loading latency, a more realistic one, was measured. Because in STR-tree loading, for reading/writing temporary files during external sorting of the dataset by *x*- and *y*-coordinates, operating system buffers are used instead of the library's buffer pool. This prevents us from calculating the exact disk IO during loading.

As the graph shows, the increase in loading latency of both trees is almost linear, although the R*-tree loads almost ~44 times slower. While the tree height was three for all datasets, the space efficiency is measured as ~70% for R*-tree and ~98% for STR-tree. In addition to that, we may compare the leaf MBRs of the R*-tree and STR-tree loaded with the 50K square objects illustrated in the Figure 3A,B below. Here, as expected, the STR-tree is much more organized than the R*-tree, and thereby the amount of intersection is less. Similar situations have been measured in other datasets. In this way, the query efficiency of the STR-tree is expected to be much better than its dynamic counterpart. Our experiments regarding query efficiency in the following subsection confirm this prediction.

Although proximity-to-square is reported as a critical parameter for the tree quality, as the *AQAR* of the workload consisting of window queries changes, we may foresee that the efficiency of the STR-tree dramatically changes. In specific, let *Q* be a workload containing a set of window queries q_i, $1 \le i \le n$ and q_{ix} and q_{iy} are the projection length of *i*th query at dimension *x* and *y*, respectively. Therefore, *AQAR* can be computed as the ratio of average projection lengths of *Q* at each dimension: that is, $AQAR = (1/n) * \left(\sum_i q_{ix} / \sum_i q_{iy}\right)$. The more *AQAR*-value differs from 1, and query performance is negatively affected. Therefore, our intuition is that stripe (or parcel) shapes tuned according to the *AQAR*-value rather than blindly applying near-square node-MBRs would increase the query efficiency of the index. Let the total number of $p * p$ parcels remain the same, instead of forming *p* groups on both axes; space can be organized such that the number of stripes on the *x*-axis is $p/AQAR$ and the number of groups (nodes) in each stripe along the *y*-axis is $p*AQAR$. We applied this refactoring within the STR-tree implementation and generated the leaf-MBRs of our new STR-tree named *Adaptive STR-tree* (Adp-STR). Figure 3C shows the new organization loaded with 50K objects where the stripe shapes are adapted to *AQAR*-value, specifically 0.3. Besides, this reorganization does not incur any additional computation during the loading process since we presumably have *AQAR*-value of the ready workload in our off-line experimental set-up.

4.2 | Querying

To validate our previous intuitions about achieving better execution time, workloads having 100 window queries each was run on three trees (R*-tree, STR-tree, and Adp-STR-tree), each of which is loaded with 50K points. While the area and *QAR* of each window query remain stable in a workload, different workloads with different *AQAR*-values of "0.1, 0.3, 0.7, 1.0, 1.4, 3.3, and 10" have been generated.

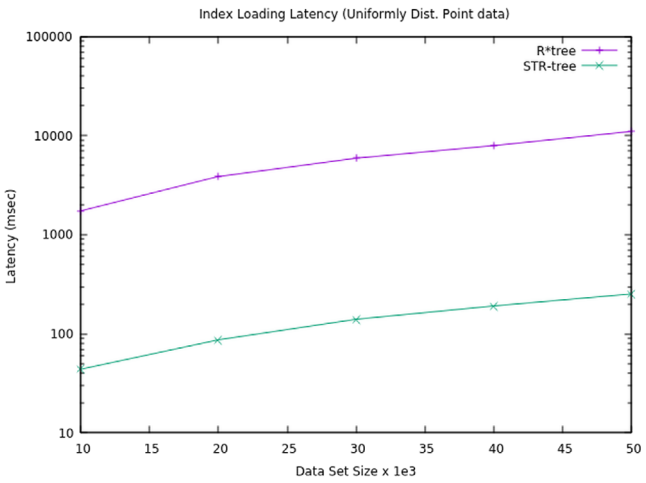


FIGURE 2 Comparison of static versus dynamic R-tree loading latency

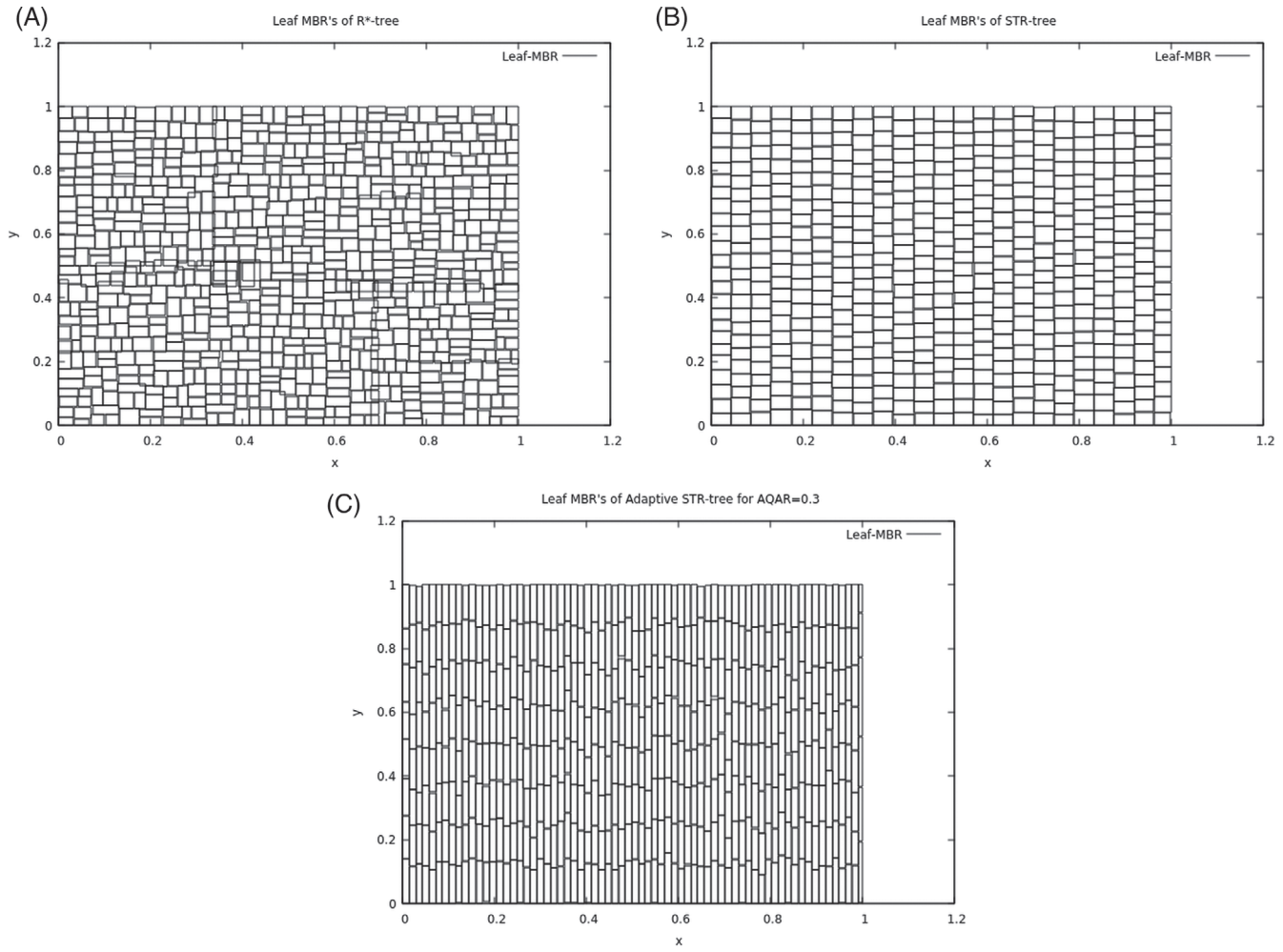


FIGURE 3 (A) Illustration of R*-tree's Leaf MBRs holding 50,000 points. (B) Illustration of STR-tree's Leaf MBRs of STR-tree holding 50,000 points. (C) Illustration of Adp-STR's Leaf MBRs of Adaptive STR-tree holding 50,000 points

The "number of nodes read" in the tree instead of the "total delay" seems a more steady parameter to evaluate the query execution efficiency. Because in the buffer pool of the library, the victim is randomly selected, disregarding tree hierarchy. This apparently affects the accuracy of query efficiency comparisons. Nevertheless, we made evaluations according to both criteria. As seen in Figure 4A, the query efficiency of the R*-tree is always lower than the STR-tree, while the Adp-STR consistently outperforms the others. As expected, both trees (R* and STR-tree) are always adversely affected by the *AQAR*-value change of the workload. For example, for a workload containing queries with *AQAR* = 0.1 and an area of 81e-4, we measured an almost 16% lower node reads in Adp-STR compared to the original STR tree.

Additionally, Figure 4B shows that the benefit of Adp-STR in terms of latency has always been preserved as the dataset volume increases. Lastly, Figure 4C illustrates the measurement of the Adp-STR's efficiency in terms of latency as a function of *AQAR* value for a specific "data volume" and "Query Area" values. We observed similar unstable latency values with other "data volume" and "Query Area" combinations. Nevertheless, we consistently obtained better latency values for Adp-STR, especially towards the extreme values of *AQAR*.

4.3 | Discussion and future work

While it seems like a good idea to adapt the tree's internal structure to the *AQAR* of the workload, some points need improvement with more careful analysis. *First*, as seen in Figure 4A,C, the efficiency of Adp-STR with changing *AQAR* is not stable. Moreover, Adp-STR sometimes exhibits worse latency performance than others (i.e., observed for *AQAR* values as seen in Figure 4C). *Second*, both data points and query windows in our execution environment have been generated uniformly in terms of their locations, which is hardly realistic. Our other unrealistic assumption is that we

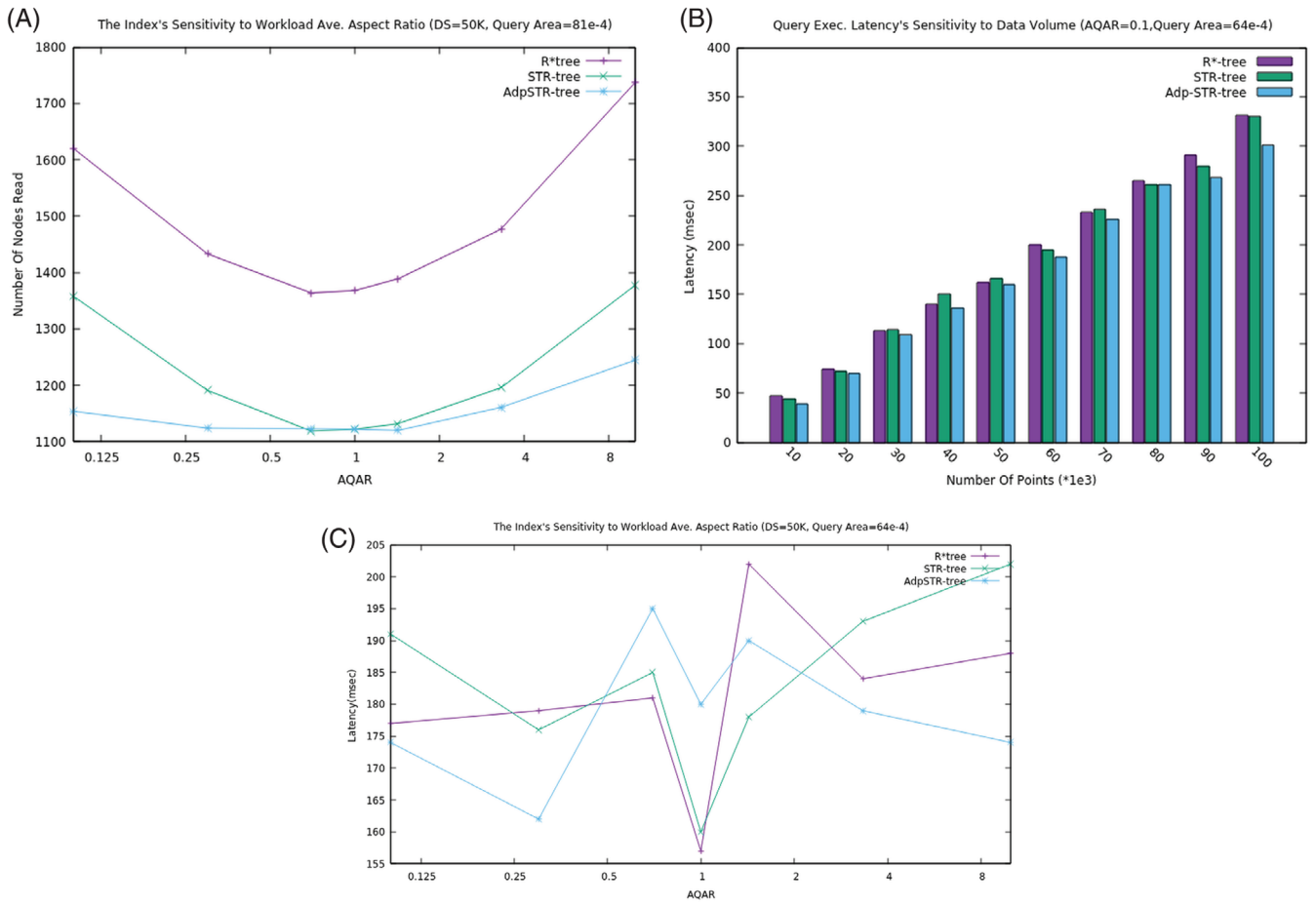


FIGURE 4 (A) The number of nodes read as a function of average query aspect ratio (*AQAR*). (B) Query execution latency as a function of data volume. (C) Query execution latency as a function of *AQAR*

generated workloads having a single type of query window, that is, having the same area and same QAR value. *Finally*, datasets with different geometries, such as rectangles, require consideration of other criteria, such as overlap.

We first discuss the first two points. There may be several reasons for the observed instability. *First*, remember that in Adp-STR-tree, we compromised proximity-to-square property (i.e., that is reported as an effective parameter for a better tree structure) for the sake of adapting to workload. On the other hand, the severe instability observed in Figure 4C compared to Figure 4A most probably depends on the victim selection policy of the library's buffer pool which is currently somewhat naïve. *Second*, in our approach, we proportionally changed the shape of all stripes based on a single value, *AQAR*, without regarding the stripe's positions and widths across the *x*-axis. However, stripe widths are also (inversely) proportional to data volume. The new stripe areas may not always be better for the current query areas in our preliminary method. More importantly, this problem is more evident with non-uniformly distributed datasets. We illustrated these effects in Figure 5A–C. It seems that the modified stripe shapes of Adp-STR holding 10K points in Figure 5A better match the queries than those of Adp-STR holding 50K points in Figure 5B. In other words, we should not have reduced the stripe width that much in Adp-STR, which contains 50K points. The rationale behind this observation is that while two contiguous parcels in a stripe are sibling disk pages (i.e., most likely physically contiguous), two parcels adjacent to each other along the *x*-axis (i.e., located in different stripes) are physically non-contiguous disk pages and requires additional random access during execution.

Figure 5C illustrates the leaf MBRs of the Adp-STR-tree holding 50K points that have non-uniformly distributed locations. In specific, we generated skewed groups at two crosswise quadrants of the unit area, thereby shrinking both stripe widths and parcel heights at skewed areas naturally. STR-tree packing strategy already reflects non-uniform data distributions (in terms of both their locations and areas) into the structure by changing the stripe (and parcel) shapes at both axes. That is to say, the stripe positions on the *x* and *y*-axis are directly dependent on the data distribution, as (in our experiments) we sort the dataset by *x* and then *y* dimensions and then pack them into nodes. However, with our first method on non-uniformly distributed datasets this time, all stripes' widths in skewed and sparse quadrants proportionally changed based on a single value, *AQAR*, without regarding the stripe's widths across the *x*-axis. As a result, this caused the benefit of our refactoring to be unstable across the entire data domain.

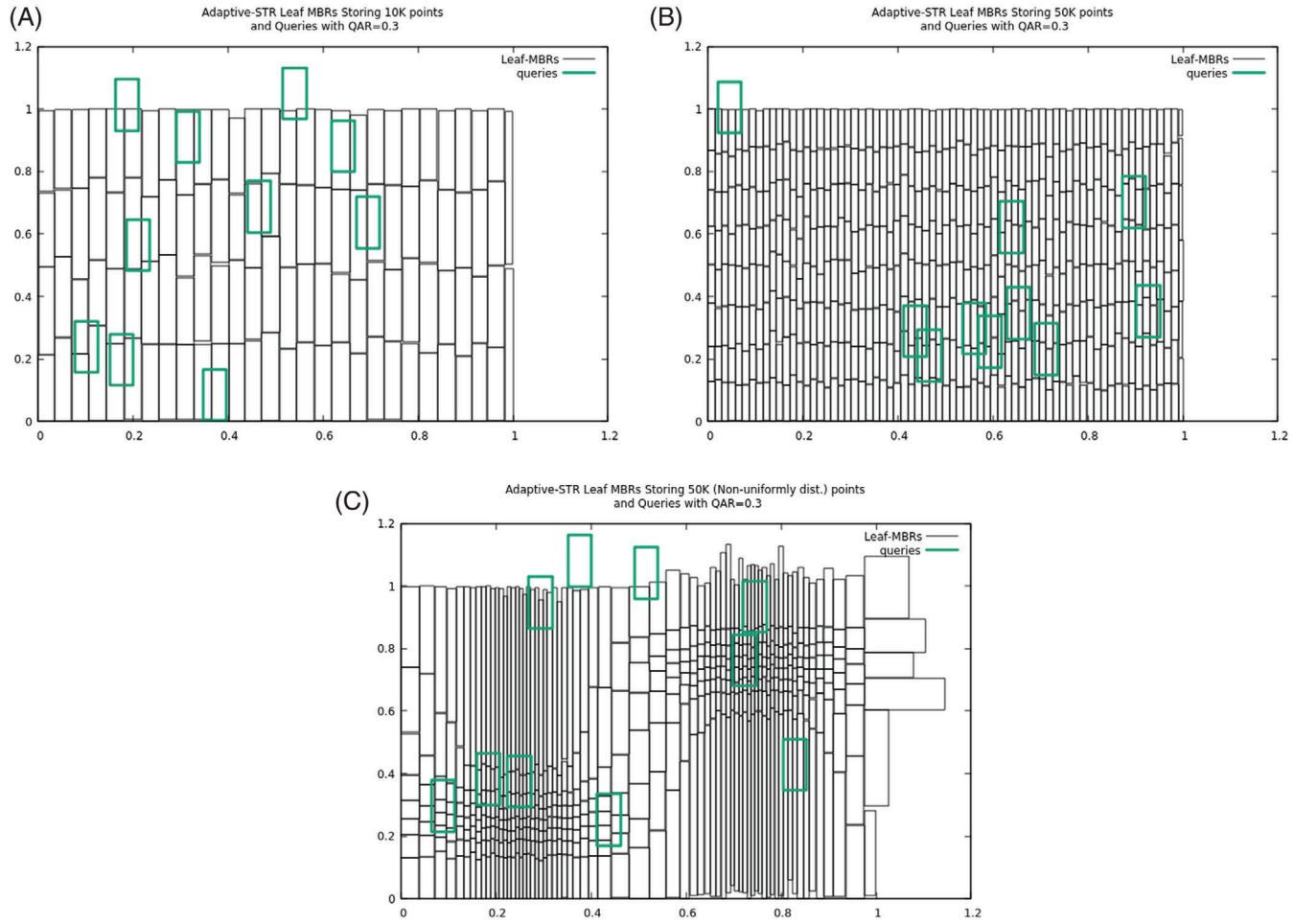


FIGURE 5 A sample workload over Leaf MBRs of Adp-STR holding (A) uniformly distributed 10K points, (B) uniformly distributed 50K points, (C) non-uniformly distributed 50K points

Similar comments are possible in workloads having non-uniformly distributed queries and/or queries with different *QAR* values. For example, suppose we either increase the stripe widths at the areas that are densely queried or even combine the adjacent stripes. In that case, the possibility of limiting the search in sibling nodes (i.e., by preventing jumping to other stripes) may increase.

The last point on our deficiency list is about spatial data having an extent. STR-tree (and other R-tree variants) are already designed for all kinds of spatial data. We run all previous experiments on datasets having square objects and obtained similar graphs. However, when we increased the average area of square objects in the dataset, the sensitivity curve of Adp-STR (at Figure 4A) got closer to STR. Because as the data objects' area increases, the overlap areas of the parcels at each level increase. This phenomenon is represented in Figure 6 with two consecutive parcels over the *y*-axis. In the figure, while *mbr1* determines the upper boundary of the parcel below, *mbr2*, which is *mbr1's* neighbor in the order on the *y*-axis, determines the lower boundary of the upper parcel. For a dataset containing larger squares, the consecutive parcels' overlap area

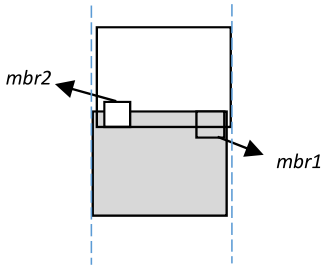


FIGURE 6 The inevitable overlap problem

increases and thereby the possibility of additional disk IO. Although this overlap problem is also an inevitable problem of R-tree variants, our suggested approach can be improved based on minimizing the overlap area between subsequent stripes rather than adjacent parcels within the same stripe.

Based on the previous observations, we may conclude that the most efficient parcel distribution does not depend solely on a single overall value, *AQAR*. Instead, similar heuristics can be applied at a "finer level of granularity," such as a stripe. With the following *stripe-based heuristics*, we expect to inject the area, the distribution, and *QAR* characteristics of a workload into the STR-tree with preserving STR-tree basics and without sacrificing short STR loading time.

Heuristic 1. Brief stats collected from the current stripe provide a hint for better shaping of the next stripe. As the stripes are already formed along the *x*-axis and sequentially loaded in the pages, quick stats may be easily collected from the current stripe. Thus the first stripe is ordinarily shaped and but the following stripes are shaped with stronger intuitions. This heuristic is based on the assumption that there are no sharp differences between the statistics in the adjacent stripes, which is much more realistic.

Heuristic 2. Let D_o and D_q be the density of "data objects" (having point or rectangle type) and "window queries" in the current stripe, respectively. More specifically, $D_o = (\text{total\#of data objects intersecting with current stripe/total area of current stripe})$ $D_q = (\text{total\#of queries intersecting with current stripe/total area of current stripe})$

can be calculated in a normalized manner. The ratio of these densities named as $\alpha = D_o/D_q$ is important. While high values of α, a signal for *sparsely* queried areas affect in the direction of the narrowing to the next stripe, oppositely, low values of α, a signal for *densely* queried areas affect in the direction of the enlargement to the next stripe.

Heuristic 3. Let $q_x = (1/n) * \sum_i q_{ix}$ be average projection length of *i* "queries" at dimension *x* in the current stripe. Similarly, according to the degree that q_x is far from "the width of the current stripe" we may expand/shrink the next stripe to better match the query subset in the next stripe.

Heuristic 4 (for point datasets). Let $AQAR_{st}$ be the *A*verage *Q*uery *A*spect *R*atio of the "window queries" in the current stripe. Obviously, according to the degree that $AQAR_{st}$ is far from 1, we may expand the next stripe for $AQAR_{st}>1$ and shrink it for $AQAR_{st}<1$.

Heuristic 5 (for rectangle datasets). In addition to $AQAR_{st}$, let $ADAR_{st}$ be the *A*verage *D*ata *A*spect *R*atio (DAR) of the "data rectangles" in the current stripe. The ratio of these values named as $\beta = AQAR_{st}/ADAR_{st}$ is also important. According to the degree that β is far from 1, we may expand the next stripe for $\beta > 1$, however for $\beta < 1$, we would better *not shrink* because if we did, wide data objects having high *DAR* value would cover more stripes.

As usual in spatial data structure design, some heuristics may conflict. Yet, we may make use of predefined thresholds at each heuristics to obtain a weighted sum value and to make a decision on the width of the next stripe. We do not think that thresholds in these heuristics need to change for different workloads. For example, queries having "large-area" and "high-*AQAR*" in "densely queried areas" will cause significant enlargement in the related stripes. To implement these heuristics, it would be better to sort the queries on the *x*-axis, which is always possible in the main memory for usual workloads. Additionally, we have to make some minor calculations (i.e., α, $AQAR_{st}$, $ADAR_{st}$, β calculations are really lightweight) at the end of each stripe during loading. We postpone the determination of the mentioned thresholds and related experiments to another study in the future. We believe that with these lightweight stats on the basis of stripes we will obtain more stable and fine-tuned STR-tree structures with only a negligible rise in loading latency.

Yet another fundamentally different approach, instead of hand-crafted heuristic rules as mentioned above, machine learning models, specifically reinforcement learning (RL)-based models, may be applied to decide the stripe/parcel positions. A very recent novel approach on the traditional R*tree, named Reinforcement Learning R-tree,[41] seems like a promotive study in the course of replacement of classic (spatial) index heuristics.

5 | OUR INCREMENTAL INDEX DESIGN METHODOLOGY ON SPATIAL DATASETS

It is apparent that traditional R-tree variants (i.e., R*tree and STR-tree) have an all-or-nothing paradigm and apply many complex heuristics to achieve a fully equipped (perfect) structure (especially the R*-tree). However, as pointed out previously, there is no need to organize "all dataset" perfectly in one time because it takes too much time. So, in this study, we used two less complex strategies to alleviate the high loading latency problem:

- SFC: Z-curve codes[42]
- B-tree: Partitioned B-tree[35]

SFC organizations provide a sequential traversing over the space with different fractals with self-similar patterns at different scales. The traversal patterns are formed by shuffling the binary coordinate values of the grid points with various patterns. Depending on the expected resolution of the space, once each dimension is represented by *k* bits, and thus a grid-point in *d*-dim space is represented by $k * d$ bits; eventually, $2^{(k*d)}$ grid points in space can be "virtually" sorted.

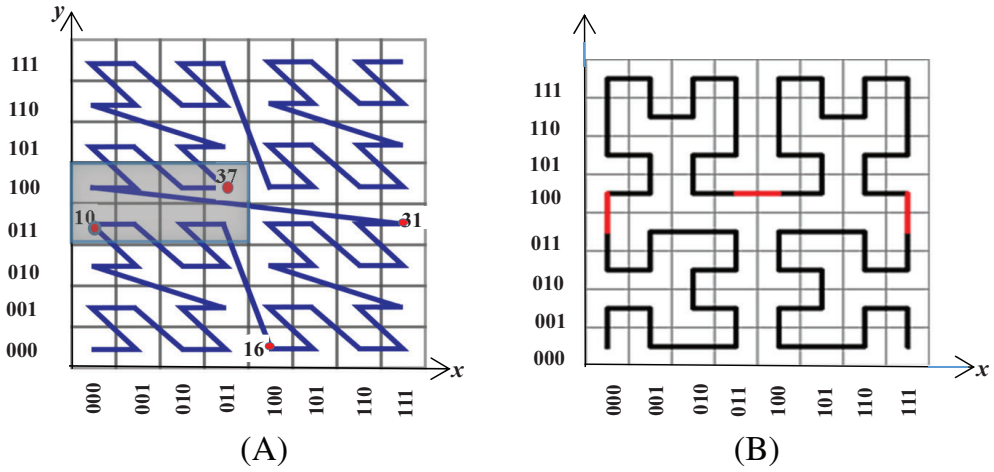


FIGURE 7 Space filling curves with (A) Z-curve (B) Hilbert-curve[40]

Figure 7 shows the Z-curve and Hilbert-curve of 64 grid points (cells) in two-dimensional space using $k = 3$-bits for each dimension. k denotes the "order of curve," which means the number of times space has been partitioned into quadrants. Linear increasing Z-values in Figure 7 are obtained by interleaving the coordinate values y and x, respectively. For example, the Z-value of the cell $(y, x) = (011,000)$ is computed as ten ("10") by bit interleaving. We need higher orders to distinguish a large number of objects with higher precision, and typically an order of 16 (i.e., 32-bit Z-codes) is sufficient since this means 2^{32} grid cells in a two-dim space. On the other hand, the principal problem of Z-curve organization is the sharp transitions between some grids. To alleviate this problem, Hilbert-curve virtually sorts the space with more smooth transitions by applying a more complex interleaving process.[42,43]

Note that we chose the lightly equipped versions of each strategy: less complex Z-curve compared to Hilbert-curve and semi-organized Partitioned-B-tree compared to fully sorted B-tree. With the first strategy, we aimed to get the "big picture" of the whole dataset, and with the second structure, we may store and search on this big picture. When we scrutinize the internal building blocks of these structures, we observe that both expose adaptable behavior. For example, we may represent the dimensions with a number of bits depending on the expected resolution of the big picture, and we may tune the batch sizes (thus the number of partitions) depending on the expected latency for loading.

5.1 | Data loading

To observe the benefits of our approach compared to STR-tree loading, we used "libmorton" library[16] to generate Morton codes *(synonym of Z-curve code)* and BDB library[15] to store codes in a Partitioned-B-tree. Our experiments used two-dimensional point datasets that are synthetically generated beforehand and are uniformly distributed within the unit area. Datasets with different sizes between (1e4-1e8) have been loaded into clustered "Partitioned B-tree" and "STR-tree," resulting in a total size of about 4 and 4.7 GB, respectively, for the largest dataset. Depending on the predefined batch size, our test program written in C++ sequentially reads subsets of data into memory and sorts the set in memory by their 32-bit Morton codes generated on-the-fly and then adds artificial leading key and finally load the set into BDB's B-tree with its bulk-loading utility. Thus, we built "clustered composite B-tree" file organized based on "*batch-id*, *Morton-code*" key and holding the object's id and the object's spatial information in the leaves of the tree. As can be seen in Figure 8, the optimum batch size*(bs)* is obtained through a parameter sweep on a dataset containing 1e6 objects. The graph shows that for batch sizes having higher than 30,000 points, in-memory sorting starts to dominate in the loading process and adversely affects the performance. This observation is orthogonal to the dataset size. Therefore, we set the optimum batch size at $bs = 30{,}000$ points in our tests to exploit BDB's bulk-loading utility as much as possible.

On the other hand, we configured STR-tree (as the structure to compare) with its optimum space efficiency (0.99) as before. Although our Linux system utilizes 4 KB pages, we set the node capacity of both structures as 32 KB to alleviate the high latency. Since sequential access is the dominant factor in bulk loading performance, large capacity is expected to lessen the latency. Lastly, both structures, BDB and STR-tree, have been configured to use as high main memory as possible. For the sake of fairness, we configured each structure to utilize 1.7 GB of main memory for their data preparation sub-processes. For example, BDB uses this memory to accumulate and sort the batch entries and then store the batch into the disk at one time. STR-tree utilizes this memory at the split and merges sub-processes of external sorting to generate the parcels from the dataset.

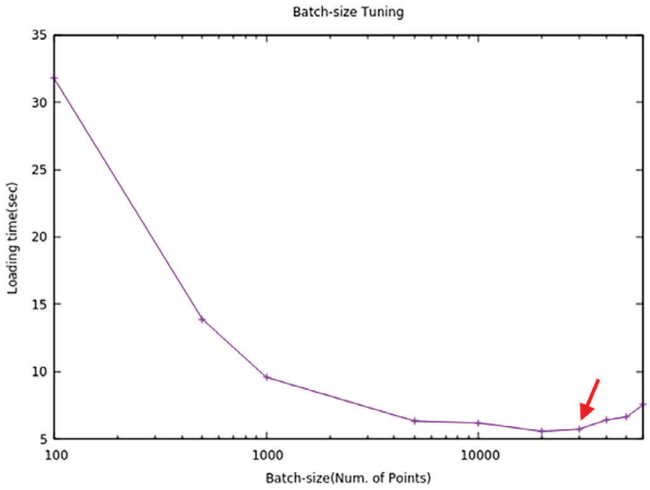


FIGURE 8 Loading delay as a function of batch-size, *bs* (for 1e6-size-dataset)

5.2 | Querying

As our primary goal in this section, to evaluate the "data-to-insight" latency, we immediately run a single-window query on the Partitioned-B-tree and STR-tree structure. The last line in our data files has a "window query" randomly generated within the unit area based on uniform distribution at both dimensions. While the window query execution is straightforward with the spatial index library's query execution utilities, running the window query on the partition-B-tree incurs additional transformations and designing some execution strategies. Unlike a single code representing a multidimensional point, we have to use several codes for a region in space. "Libmorton" library generates 32 or 64 bit Morton codes only for points in a spatial area. An arbitrary window region's Morton codes are not as simple as taking all between the minimum and maximum code corresponding to the lower-left and upper-right corner of the window. This phenomenon can be observed in Figure 7A, with a window query having the Z-code values of 10 and 37 for its corners. Note that not all values in the range (10–37) falls into the region; instead, we need to find the list of sub-ranges ("10-11, 14-15, 32-33, and 36-37" for our example) that pass through the query region.

There have been a number of studies on finding the list of sub-range.[44–46] For example, Ramsak et al.[44] uses two functions *getNextZvalue* and *getNextZvalueExit* to find the sub-ranges (segments) in the list. The first function finds the closest (minimum) Z-value from a point outside the window to a point inside, while the other function works the other way around. Lawder and King[45] apply a similar technique on Hilbert–curves with a single function this time. In general, both algorithms start with finding the corner that has the minimum code value using the B-tree search utility and then use the smart jumping iteratively throughout the leaves (i.e., to result with as few disk IO as possible) until reaching the maximum code of the window. "Smart jumps" that start searching a new sub-range can be done by traversing some internal (index) nodes instead of start searching at the root for each sub-range. For deep B-trees (i.e., larger trees) when the internal nodes cannot be cached completely, this technique improves the efficiency of the execution.

Alternatively, Aboulnaga and Aref[46] adapt the Quad-tree technique to find sub-ranges all at once without accessing the B+-tree index. The Quad-tree, a hierarchical data structure based on recursively decomposing the space into quadrants, is an efficient structure to represent regions within a space. Space intersecting with the region is decomposed into quadrants until one or more new quadrants are entirely contained by the region. Thus, quadrants that are not intersecting with the region stay remain intact. Encoding quadrants at different levels as Morton codes enables regions (queries) to be represented efficiently, thereby finding the list of sub-ranges. Figure 9 shows the required decomposition and its representative Quad-tree of the window query used in Figure 7. Based on the Z-curve pattern in this organization, if the SW quadrant is coded as 00, SE(01), NW (10), and NE (11), then the window region's Morton codes become 001010(**10**), 001011(**11**), 001110(**14**), 001111(**15**), 100000(**32**), 100001(**33**), 100100(**36**), and 100101(**37**) which generates sub-ranges of ("**10-11**, **14-15**, **32-33** and **36-37**").

We adopted the Quad-tree technique and implemented "*FindingSubRanges*" function in the window query execution part of our evaluations. (see Appendix A.1 for the pseudocode) The main reason is its simplicity compared to functions *(getNextZvalue* and *getNextZvalueExit*) previously discussed. Additionally, implementing "smart jumps" incurring modifications of the BDB's B-tree kernel increases the complexity. On the other hand, the Quad-tree technique controls the decompositions with efficient bitwise operations without any disk access. We may also apply the "order of Quad-tree" as a tuning parameter for a trade-off between accuracy and the query execution time. Table 1 briefly gives this trade-off for both the "query execution" part and the "loading 1e7-dataset" part of our methodology.

We observed that with the order of 16, the "*FindingSubRanges*" function generates almost $L = 50{,}000$ segments with a latency (0.14 s). When we decrease the precision by applying a smaller order (i.e., 8), the sub-ranges generation performed almost 400 times faster, losing only 2% accuracy.

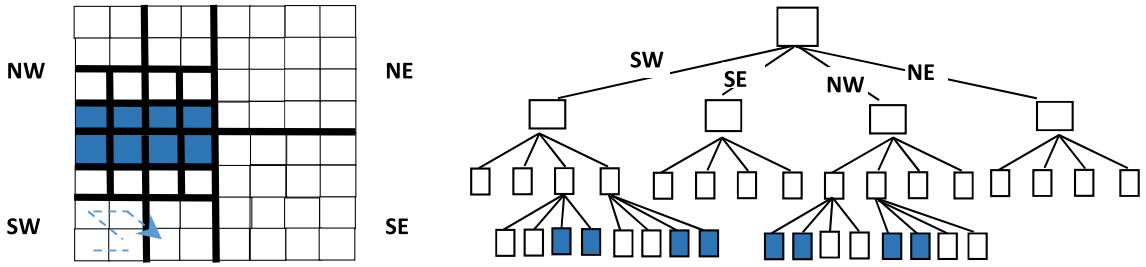


FIGURE 9 Space decompositions in Quad-tree and its tree representation

TABLE 1 Trade-off between latency and accuracy versus Order of Morton codes

	Window query			Point dataset	
Order	Num. of sub-ranges (*L*)	Latency for *FindingSubRanges* (s)	Accuracy of ResultSet	Loading latency (s)	Duplicate entries
16	~50,000	0.14	99.999%	~55	0.001%
8	125	~0.0003	98%	~55	20%

When the order is 16, $L = 50{,}000$ segments are mostly points (i.e., grid-cells, but not sub-ranges) corresponding to the window's borders, and only about 125 comes from the larger grid-cells within the window region. Thus we only lose data objects on these border points (i.e., only the 2% of all result set) when we reduce the precision. On the other side, utilizing lower-order Morton codes for data points caused a significant number of duplicate entries into B-tree without any gain on loading latency, as shown in the last column of Table 1. As a result, we can reduce the Morton order in mapping the query window (especially for subsequent queries in the workload). Still, it would be better to use better quality mappings (order of 16) for data points.

After generating sub-ranges, another critical point in our design has been to search these sub-ranges within "each partition" in the "partitioned-B-tree," which is a little different from searching all sub-ranges within a traditional B-tree with a single partition. On the one hand, we may begin the search from the root for each sub-range and each partition; on the other hand, we may always continue searching throughout the tree's leaves without any search from the root. The tradeoff depends on several parameters like the height of the tree (i.e., dataset size), the size of sub-ranges set (i.e., query's spatial characteristics), cache size (i.e., the percentage of B-tree nodes cached), and disk page size. For example, large disk pages yield high scan bandwidths; thus, end-to-end index scan becomes more attractive.

On the contrary, small window queries generate a small set of sub-ranges, and an end-to-end index scan would unnecessarily access many leaf pages; thus, B-tree search for each sub-range seems more attractive. We opted for a moderate approach and started a new search from the root only for "each partition." Since the sub-range set is naturally generated sorted by the Morton code and codes in the B-tree leaves are already stored in sorted order in each partition, we devised the search process with an algorithm like "finding common elements of two sorted lists". However, this time we need a nested-loop incurring a quadratic run-time since this process will repeat for each distinct partition. The whole algorithm is given in Appendix A.2. We chose the B-tree leaves as the outer loop since this results in scanning B-tree leaves (in a partition) in one-pass and thus significantly reduces the random Disk IO. The algorithm starts finding the left-most segment in each partition with a B-tree search and sequentially scans two lists until the end of B-tree leaves in the partition or the sub-range list.

The algorithmic time complexity can be defined as $O((B/|bs|) * D * \log_{fo} B + (B/|bs|) * (bs + SRs))$, where D denotes the time for a single Disk IO, B denotes the total number of leaf nodes, fo means the average fan-out of the tree, and $|bs|$ is the batch size in terms of page. Actually, $(B/|bs|)$ is the number of partitions. Lastly, as defined previously, bs is the batch size, and SRs is the sub-range set size, both in terms of the number of records (Z-values). If we assume the number of partitions in the tree as a small constant compared to two set's sizes, we may hope the run-time converges to linear run-time.

5.3 | Obtaining better data-to-insight latency

We evaluated the data-to-insight latency on point datasets in different sizes based on design decisions on data loading, and query execution mentioned previously. For brevity, we present results on five datasets in a table. Table 2 gives the total data-to-insight latency versus dataset sizes ranging from 1e4 to 1e8. Table, specifically, shows two consequent stages of data-to-insight latency with two leading columns (i.e., loading and execution stages). The first important observation is that the impact of our index design appears on datasets larger than 1e6. Data sets observed an

TABLE 2 Comparison of data-to-insight latency for STR-tree and Partitioned-B-tree

	STR-tree			Partitioned-B-tree		
Dataset size	Loading latency (s)	First query's execution time (s)	Data-to-insight (s)	Loading latency (s)	First query's execution time (s)	Data-to-insight (s)
1e4	0.09	0.01	0.1	0.74	0.15	0.89
1e5	0.67	0.06	0.73	1.17	0.26	1.43
1e6	6.69	0.70	**7.39**	5.92	1.15	**7.07**
1e7	83.20	9.77	**92.97**	55.21	11.12	**66.13**
1e8	1173.29	70.37	**1243.66**	693.34	130.5	**823.84**

Note: Data sets observed an improvement in "loading latency" and "data-to-insight latency" are colored in the table, respectively shaded and bold formatted.

improvement in "loading latency" and "data-to-insight latency" are colored in the table, respectively shaded and bold formatted. We observed that point data mappings on linear ordering have been effective in Partitioned-B-tree's loading process for small-size datasets. However, this adverse effect disappears with increasing dataset sizes. Besides, the gap between loading (thus data-to-insight) latencies of two structures is growing as the data size increases. While Partitioned-B-tree is loaded with 12% better latency at modest dataset sizes (i.e., 1e6), the benefit reaches 41% at the largest dataset.

After loading, the attached latency value comes from the window query. As expected, query execution delay on superficially loaded Partitioned-B-tree is always higher than perfectly packed STR-tree. Even the gap between execution latencies reaches almost two times at the largest dataset (i.e., 1e8). However, we obtained better values with our proposed index loading methodology for larger-scale spatial datasets in terms of data-to-insight latency. As can be seen from Table 2, we reach about 34% better data-to-insight latency with our new design for the largest dataset.

The query execution part needs some further attention. There are two essential burdens within this operation: "mapping of window region to sub-ranges" and "utilizing B-tree search facilities." We observed that the first part, the window region's mapping, consistently takes almost 0.14 s, so that it becomes negligible with increasing dataset sizes. For example, it takes only 0.1% of the whole execution process in the test on the largest dataset. Thus, the burden almost totally comes from the B-tree utilization. The fact that the searched sub-ranges are disseminated throughout the partitions of the Partitioned-B-tree increases this burden inevitably. As pointed out previously, we only used the B-tree search starting from the root for only each partition. However, this is not the dominant factor in this burden since we already built a shallow B-tree as much as possible by using large-size data pages (i.e., 32K). The Partitioned B-tree built for the largest dataset (i.e., 1e8) has a size-on-disk of almost 4 GB and a height of only 3, and BDB's cache mechanism almost always caches the internal pages. Therefore we may deduce that the execution time of the first query is highly dependent on disk scan bandwidth.

Last but not least, data distribution is a parameter that may change the quality of an index. As noted beforehand, our spatial data are uniformly distributed points within the unit area; that is, our design is a Point Access Method that is incrementally loaded. On the other hand, non-point objects having spatial extent expose different complexities in data loading and querying due to the overlap of geometries.[20] Therefore, the quality of spatial access methods that are indexing spatial data having extent are more sensible to non-uniform data distribution due to more overlap problem of geometries. Based on this intuition, if we run our experiments with more skewed data, we may foresee that STR-tree's efficiency (especially the query execution performance) degrades due to the more overlaps of "internal node MBRs." This degradation has been shown in many experimental studies.[12,28,32] There would be no overlap throughout the leaves storing groups of points, but an increase in overlaps in uppers nodes will be enough for such degradation.

On the other hand, our method, which is based on mapping two-dim points into SFC codes, will be safe from this degradation both at the data loading and the query execution stages as far as we generate different codes for each point in the dataset. We previously achieved this with a suitable resolution level. (For our largest dataset, an order of 16 has been sufficient to generate SFC codes without any duplication.). Consequently, our design to achieve better data-to-insight delay is orthogonal to data distribution, and this seems to be another promotive outcome of our study.

5.4 | Discussion and future work

In this part of our study, we aimed to achieve better data-to-insight latency and achieved our goal in large datasets. Yet, the continuous use of the Partitioned-B-tree in its current form may not be suitable for other queries that will come in the workload. We may roughly estimate that only the next eight queries would benefit from our design. As we discussed previously, we may decrease the quality of service (*QoS*) by generating

sub-ranges with lower precision. In Table 1, we see that the query execution performed almost 400 times faster with losing only 2% of accuracy. However, because the execution delay mostly comes from the B-tree utilization, not the window mappings mentioned beforehand, lowered *QoS* does not help.

It is already expected in incremental index design that the index needs to be improved (i.e., reorganized) for more efficient usage in the subsequent queries. There seem to be several options to incrementally tune the Partition-B-tree to get the most benefit from the index in our design. The first idea is to stay on the same domain and incrementally transform the Partitioned B-tree into a conventional B-tree (with a single partition). As proposed in its original article, the upcoming searches on this transient index make this reorganization during execution (in a by-product manner). The burden of the continuous scan of leaf nodes will decrease as long as query executions collect searched data from partitions and cause them to shrink or eventually disappear. Our second idea is to load a "spatial structure" incrementally with a batch of spatially related data objects. We can get the benefit of the spatially related property of the objects in each query's result set. The most typical queries like window and k-NN queries collect objects that are naturally spatially related to each other. These subsets can be "bulk-inserted" into the R*-tree with better space utilization, although it may not be as good as bulk-loading. We mentioned some benefits of "bulk insertion" in Section 2.1. The techniques used in bulk insertion may be applied in our methodology. (One problem in this approach is that the size of the result set can be arbitrary. Some heuristics can be proposed for the problem. For example, if it is too tiny, bulk insertion may be canceled or postponed, and if it is too large, we need to partition the result set following the capacity of the R-tree.) Another critical point in this idea is that the Partitioned-B-tree gradually shrinks with bulk deletions, and the R*tree slowly grows with bulk-insertions. There are some subtle points in this methodology that are needed to be discussed further.

As can be predicted, assessing such highly dynamic organizations' space and execution efficiency requires defining additional criteria, some of which we mentioned in the Literature Review section (such as the cumulative execution time and convergence time). Since the index construction is distributed across the workload's lifetime, the incremental index's benefits depend on how the reorganizations are performed and scheduled during the workload span.[39] For example, workload characteristics can take many different forms, and workloads containing completely random queries will result in the longest possible stages in the adaptation procedure. Workloads having more uniform features would ease the assessment procedure. In our future research, we plan to implement and assess the benefits of our ideas on the incremental index in terms of these novel quality criteria.

6 | CONCLUSION

Our study mentions current data access characteristics of emerging technologies, such as social media applications and exploratory data analysis environments. We pointed out the need for novel physical data organizations for those technologies, particularly the need for novel index construction methodologies. More specifically, in this article, we studied adaptive bulk-loading approaches on large-scale spatial datasets. Our study is twofold. One is to bulk-load the R-tree structure by considering the average aspect ratio of a ready workload. The other is to bulk-load the large spatial dataset more superficially to obtain a better data-to-insight latency. We believe that these two concrete contributions within each sub-problem may be a baseline for future work on this area. Spatial databases aside, there have been many studies addressing the indexing of temporal data with R-tree-based index structures.[47,48] Hence, our workload-aware and incremental approaches may also be applied in different multi-dimensional domains, that is, temporal databases, spatio-temporal databases, and even some use cases of data provenance management. For example, some studies[49,50] focus on managing provenance metadata, which is primarily time series-based data, better to understand the history of the data and processes. Lastly, some application domains[51] using spatial big data frameworks are another candidate area that our approaches on bulk-loading can be applied.

In the first part of the study, we modified the well-known STR-tree bulk-loading organization by tuning the number of stripes and the parcel shapes within the stripes depending on the workload's aspect ratio. This intuitive and straightforward refactoring named Adaptive-STR performed about 16% better than the query execution performance of the traditional STR tree without any increase at the loading time. Additionally, we pointed out the need for more holistic approach in determining the parcel shapes (and distributions) by considering the workload and the data distribution characteristics.

In the second part of our study, we aimed to load large datasets more quickly and get the first results from the data, called data-to-insight latency, faster. We observed that similar problems had been studied in traditional databases recently. The essence of studies in this area is to compromise the benefits of "bulk-loading" to some degree by superficially loading the large data and to improve the temporary index during the workload span. In relation to that, the literature review section discussed two techniques (database cracking and adaptive merging) that load all data superficially and thus faster. These techniques essentially provide an incremental loading of the one-dimensional indexes. We adopted the Partitioned B-tree structure used in the adaptive merging method due to its applicability to disk-based data structures, like B-tree and R-tree. Our index loading methodology has also practiced SFC, specifically Z-curve (Morton codes), enabling fast mapping of spatial objects to linear ordering. Thus we built a "clustered-composite B-tree index" to get a big picture of the large spatial dataset. To assess the benefits of our proposed method, we ran simulation tests with five datasets of different sizes. We achieved better data-to-insight latencies by a factor of 34% for the largest dataset

in our testbed. Finally, we discussed possible solutions to incrementally improve our current design for the subsequent queries in the workload and examined some criteria suitable for such dynamic physical organizations.

ACKNOWLEDGMENTS

I am grateful to Mehmet Aktaş and Kemal Bıçakçı for their contributions they have made throughout the revision process. I also wish to thank the reviewers for their comments and helpful suggestions.

DATA AVAILABILITY STATEMENT

Data available on request from the authors

ORCID

Mustafa Utku Kalay https://orcid.org/0000-0002-8002-0268

REFERENCES

1. Building the new Twitter.com. Accessed June 10, 2021. https://blog.twitter.com/engineering/en_us.html.
2. Agrawal S, Chaudhuri S, Narasayya VR. Automated selection of materialized views and indexes in SQL databases. Proceedings of the 26th International Conference on Very Large Data Bases; 2000:496–505.
3. Chaudhuri S, Narasayya VR. An efficient, cost-driven index selection tool for Microsoft SQL server. Proceedings of the 23th International Conference on Very Large Data Bases; 1997:146–155.
4. Bruno N, Chaudhuri S. An online approach to physical design tuning. *IEEE 23rd International Conference on Data Engineering*. Washington, DC: IEEE Computer Society; 2007.
5. Graefe G, Kuno H. Self-selecting, self-tuning, incrementally optimized indexes. Proceedings of the EDBT '10: 13th International Conference on Extending Database Technology; March 2010:371–381. https://doi.org/10.1145/1739041.1739087.
6. Idreos S, Kersten ML, Manegold S. Database cracking. *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research*. Vol 3; Asilomar, CA: Online Proceedings; 2007:1-8.
7. Pavlovic M, Sidlauskas D, Heinis T, Ailamaki A. QUASII: Query-Aware Spatial Incremental Index. Proceedings of the 21st International Conference on Extending DatabaseTechnology (EDBT 2018); 2018; Vienna, Austria.
8. Sanbonmatsu KY, Joseph S, Tung CS. Simulating movement of tRNA into the ribosome during decoding. *Proc Natl Acad Sci*. 2005;102(44):15854-15859.
9. Tauheed F, Biveinis L, Heinis T, Schürmann F, Markram H, Ailamaki A. Accelerating range queries for brain simulations. *ICDE*. Arlington, VA: IEEE; 2012:941-952.
10. Beckmann N, Kriegel HP, Schneider R, Seeger B. The R*-tree: an effcient and robust access method for points and rectangles. Proceedings of the SIGMOD Conference; 1990; Atlantic City, NJ:322–331.
11. Leutenegger ST, Lopez M, Edgington J. STR: a simple and efficient algorithm for R-tree packing. *ICDE*; Birmingham: IEEE; 1997:497-506.
12. García YJ, López MA, Leutenegger ST. A greedy algorithm for bulk loading R-trees. ACM-GIS. 6th International Symposium on Advances in Geographic Information Systems. November 1998, Washington D.C.
13. Zhang R, Qi J, Stradling M, Huang J. Towards a painless index for spatial objects. *ACM Trans Database Syst*. 2014;39:1-42. https://doi.org/10.1145/2629333
14. Spatial Index Library: libspatialindex. Accessed December 10, 2020. https://libspatialindex.org/.
15. Oracle Berkeley DB. Accessed December 10, 2020. https://www.oracle.com/database/technologies/related/berkeleydb.html.
16. Jeroen Baert's Blog. Libmorton: A C++ library for Morton order encoding/decoding. Accessed December 10, 2020. https://www.forceflow.be/2016/01/18/libmorton-a-library-for-morton-order-encoding-decoding/.
17. Bayer R, McCreight E. Organization and maintenance of large ordered indices. SIGFIDET '70: Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description. 1970;107-141. https://doi.org/10.1145/1734663.1734671
18. Ramakrishnan R, Gehrke J. *Database Management Systems*. 3rd ed. New York: McGraw-Hill; 2003.
19. Samet H. *Foundations of Multidimensional and Metric Data Structures*. San Francisco, CA: Morgan Kaufmann Publishers Inc.; 2005.
20. Gaede V, Guenther O, et al. Multidimensional Access Methods. *ACM Computing Surveys*. 1998;30(2):170–231. https://doi.org/10.1145/280277.280279.
21. Samet H. Spatial data structures. In: Kim W, ed. *Modern Database Systems, the Object Model, Interoperability and beyond*. Washington, DC: ACM Press and Addison-Wesley; 1995:361-385.
22. Guttman A. R-trees: a dynamic index structure for spatial searching. Proceedings of the SIGMOD Conference; 1984; Boston:47–57.
23. Sellis T, Roussopoulos N, Faloutsos C. The r+−tree: a dynamic index for multi-dimensional objects. In Stocker, P. M. and Kent, W., Eds., *Proceedings of the Thirteenth International Conference on Very Large Databases (VLDB)*, pp. 507–518. Washington, DC: VLDB Endowment Inc.; 1987
24. Manolopoulos Y, Nanopoulos A, Papadopoulos A, Theodoridis Y. *R-Trees: Theory and Applications*. London: Springer-Verlag; 2005. https://doi.org/10.1007/978-1-84628-293-5
25. Chen L, Choubey R, Rundensteiner E. Bulk-insertions info R-trees using the small-tree-large-tree approach (ACM-GIS'1998):161–162. https://doi.org/10.1145/288692.288722.
26. Choubey R, Chen L, Rundensteiner E. GBI: a generalized R-tree bulk-insertion strategy. *SSD '99: Proceedings of the 6th International Symposium on Advances in Spatial Databases*. Berlin, Heidelberg: Springer-Verlag; 1999:91–108. https://doi.org/10.1007/3-540-48482-5_8
27. Lee T, Moon B, Lee S. Bulk insertion for R-trees by seeded clustering. *Data Knowl Eng*. 2006;59:86-106. https://doi.org/10.1016/j.datak.2005.07.011
28. Kamel I, Faloutsos C. On packing R-Trees. Proceddings of the CIKM ACM; 1999. https://doi.org/10.1145/170088.170403.
29. Roussopoulos N, Leifker D. Direct spatial search on pictorial databases using packed R-trees. *Proceedings ACM SIGMOD Conference on Management of Data*; Washington, DC: ACM; 1985:17-31.
30. Arge L. The buffer tree: a new technique for optimal I/O algorithms. *Proceedings of Fourth Workshop on Algorithms and Data Structures (WADS)*. Lecture Notes in Computer Science. Vol 955. Berlin, Germany; Springer-Verlag; 1995.

31. Achakeev D, Seeger B, Widmayer P. Sort-based query-adaptive loading of R-trees. *ACM Int Conf Proc Ser*. 2012;2080-2084. https://doi.org/10.1145/2396761.2398577
32. Arge L, Berg M, Haverkort H, Yi K. The priority R-tree: a practically efficient and worst-case-optimal R-tree. *ACM Trans Algor*. 2008;4. https://doi.org/10.1145/1328911.1328920
33. Tao Y, Papadias D. Adaptive index structures. *VLDB*; Washington, DC: VLDB Endowment Inc.; 2002.
34. Becker L, Partzsch H, Vahrenhold J. Query responsive index structures. Proceedings of the GIScience '08; 2008:1–19.
35. Graefe G. Sorting and indexing with partitioned B-trees. *CIDR*. 2003.
36. Idreos S, Manegold S, Kuno H, Graefe G. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment*. Washington, DC: VLDB Endowment Inc.; 2011;9:586–597. https://doi.org/10.14778/2002938.2002944
37. Alagiannis I, Borovica R, Branco M, Idreos S, Ailamaki A. NoDB: efficient query execution on raw data files. *Communications of the ACM*. 2015;58(12):112–121.
38. Karpathiotakis M, Branco M, Alagiannis I, Ailamaki A. Adaptive query processing on RAW data. Proceedings of the VLDB Endowment. 2014;7(12):1119–1130.
39. Graefe G, Idreos S, Kuno H, Manegold S. Benchmarking adaptive indexing. *TPCTC*. 2010;6417:169-184. https://doi.org/10.1007/978-3-642-18206-8_13
40. Pavlovic M, Tzirita EZ, Sidlauskas D, Heinis T, Ailamaki A. Space odyssey: efficient exploration of scientific data. Proceedings of the ExploreDB; 2016.
41. Gu T, Feng K, Cong G, Long C, Wang Z, Wang S. The RLR-tree: a reinforcement learning based R-tree for spatial data; 2021. arXiv preprint arXiv:2103.04541.
42. Faloutsos C, Roseman S. Fractals for secondary key retrieval. *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM; 1989:247–252. https://doi.org/10.1145/73721.73746
43. Text and Images © 2001 Kerry Mitchell. https://www.fractalus.com/kerry/tutorials/hilbert/hilbert-tutorial.html.
44. Ramsak F, Markl V, Fenk R, Zirkel M, Elhardt K, Bayer R. Integrating the Ub-tree into a database system kernel. *VLDB*; Washington, DC: VLDB Endowment Inc.; 2000:263-272.
45. Lawder JK, King PJH. Querying multi-dimensional data index using the hilbert space-filling curve. SIGMOD 2001 Rec. 30, 1, 19–24.
46. Aboulnaga A, Aref WG. Window query processing in linear Quadtrees. *Distrib Parallel Databases*. 2001;10:111-126. https://doi.org/10.1023/A:1019256828670
47. Bliujutk R, Jensen CS, Saltenis S, Slivinskas G. R-tree based indexing of now-relative bitemporal data. Proceedings of the 24th VLDB Conference; 1998; New York, NY.
48. Jensen CS, Saltenis S. R-tree based indexing of general spatio-temporal data. *A timecenter technical report*. 1999.
49. Tas Y, Baeth MJ, Aktas MS. An approach to standalone provenance systems for big social provenance data. Proceedings of the 12th International Conference on Semantics, Knowledge and Grids (SKG); 2016.
50. Baeth MJ, Aktas MS. Detecting misinformation in social networks using provenance data. 2017 13th International Conference on Semantics, Knowledge and Grids (SKG), Beijing, China; 2017.
51. Eken S, Sayar A. A MapReduce-based big spatial data framework for solving the problem of covering a polygon with orthogonal rectangles. *Tehnicki Vjesnik*. 2019;26:36-42. https://doi.org/10.17559/TV-20170418094421

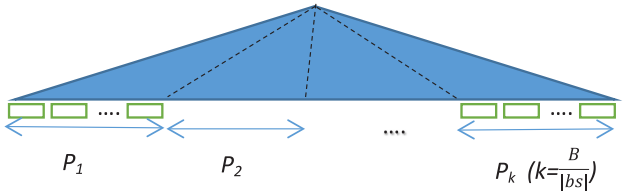
How to cite this article: Kalay MU. Novel approaches on bulk-loading of large scale spatial datasets. *Concurrency Computat Pract Exper*. 2021;e6596. https://doi.org/10.1002/cpe.6596

APPENDIX A

A.1 Pseudo-code for *FindingSubRanges* that outputs the Z-value ranges of the window query (wquery)

```
1 procedure FindingSubRanges (wquery, quadregion,order) // The function is initially called by setting
                                                        // the quadregion="unitarea" and order = 0
2 begin
3       if (wquery.CONTAINS [quadregion])
4               REPORT_SubRange (quadregion); // generate a sub-range by finding 32-bit Morton codes of
                                              //left- bottom and right- top points with libmorton library
5       else if (wquery.INTERSECTS (quadregion))
6           order: = order + 1;
7           foreach sub-quad ε quadregion do // traverse sub-quads compatible with libmorton's Z-order
                                             //s.a. SW, SE, NW and NE
8               if(order < 16)               // we generate 32 bit Morton codes
9                   FindingSubRanges (wquery, sub-quad,order)
10              else
11                  REPORT_SubRange (sub-quad);
12          enddo;
13 end;
```

A.2 A figure depicting the Partition B-tree and the pseudo-code for the complete process to obtain a data-to-insight from a raw spatial dataset



B: total number of leaf nodes
$|bs|$: batch size in terms of page
k : number of partitions

```
1 procedure FirstWindowQuery(wquery, datafile) // Execution of window query on a raw data file
2 begin
3       pbtree ← BuildPartitionBtree(datafile) //Load raw data into PartitionBtree with adding an artificial
                                                // leading key
4       SRset ← FindingSubRanges (wquery) // Generate list of subranges to represent window query
5       foreach partition ϵ pbtree do
6         sr ← getFirst_sr (SRset)
7         sr_L: left-most code; sr_R: rigth-most code
8         obj ← SearchInBtree(pbtree, sr_L, partition); // locate the first object in the current "partition"
9         REPORT (obj) //report first object if there is a MATCH
10          foreach sr ϵ SRset do
11              objs ←FindMatching_obj&sr (sr, pbtree,partition); // find matching objects within
12                                  // current "sr" by traversing B-tree leaves in current "partition"
13              REPORT (objs) //report other objects in the current partition
14          enddo;
15        enddo;
16 end;
```

PS. The loop at *Line 10* has a linear time complexity since it finds all matching objects within each subrange with only one iteration over **SRset** list and current **partition**.