Packing R-trees with Space-Filling Curves: Theoretical Optimality, Empirical Efficiency, and Bulk-loading Parallelizability

JIANZHONG QI, The University of Melbourne, Australia
YUFEI TAO, Chinese University of Hong Kong, Hong Kong, China
YANCHUAN CHANG, The University of Melbourne, Australia
RUI ZHANG, The University of Melbourne, Australia

The massive amount of data and large variety of data distributions in the big data era call for access methods that are efficient in both query processing and index management, and over both practical and worst-case workloads. To address this need, we revisit two classic multidimensional access methods – the R-tree and the space-filling curve. We propose a novel R-tree packing strategy based on space-filling curves. This strategy produces R-trees with an asymptotically optimal I/O complexity for window queries in the worst case. Experiments show that our R-trees are highly efficient in querying both real and synthetic data of different distributions. The proposed strategy is also simple to parallelize, since it relies only on sorting. We propose a parallel algorithm for R-tree bulk-loading based on the proposed packing strategy, and analyze its performance under the massively parallel communication model. To handle dynamic data updates, we further propose index update algorithms that process data insertions and deletions without compromising the optimal query I/O complexity. Experimental results confirm the effectiveness and efficiency of the proposed R-tree bulk-loading and updating algorithms over large data sets.

CCS Concepts: • **Theory of computation → Data structures and algorithms for data management**; • **Information systems → Multidimensional range search**; **Spatial-temporal systems**.

Additional Key Words and Phrases: R-trees, window queries, rank space, logarithmic method

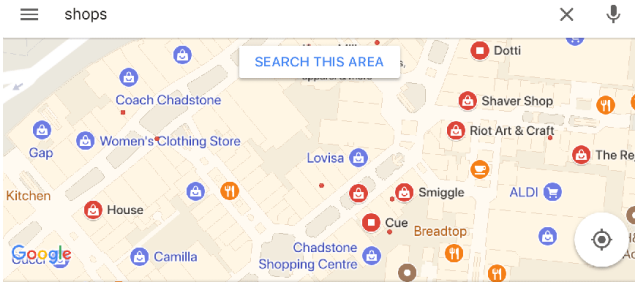
ACM Reference Format:
Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2020. Packing R-trees with Space-Filling Curves: Theoretical Optimality, Empirical Efficiency, and Bulk-loading Parallelizability. *ACM Trans. Datab. Syst.* 1, 1, Article 1 (January 2020), 45 pages. https://doi.org/10.1145/3397506

1 INTRODUCTION

Spatial databases have been traditionally used in geographic information systems, computer-aided-design, multimedia data management, and medical studies. They are becoming ubiquitous with the proliferation of location-based services such as digital mapping, augmented reality gaming, geosocial networking, and targeted advertising. For example, in digital mapping services such as Google Maps, the "search this area" functionality supports querying *places of interest* (POIs) such as shops within a given view area (cf. Fig. 1a). In a popular augmented reality game, Pokémon GO [58],

Authors' addresses: Jianzhong Qi, The University of Melbourne, Melbourne, Victoria, 3010, Australia, jianzhong.qi@unimelb.edu.au; Yufei Tao, Chinese University of Hong Kong, Shatin, Hong Kong, China, taoyf@cse.cuhk.edu.hk; Yanchuan Chang, The University of Melbourne, Melbourne, Victoria, 3010, Australia, yanchuanc@student.unimelb.edu.au; Rui Zhang, The University of Melbourne, Melbourne, Victoria, 3010, Australia, rui.zhang@unimelb.edu.au.

0362-5915/2020/1-ART1 $15.00
https://doi.org/10.1145/3397506



(a) Shops (blue and red dots) in a map view

(b) Gaming objects in a game view

Fig. 1. Window queries in real applications

every player has an avatar placed in the game map based on the player's geographical location. The players can interact with gaming objects (e.g., "pokémon") in the game view through their avatars (cf. Fig. 1b). Managing POIs or gaming objects in a view which usually has a rectangular window shape is a typical application of spatial databases.

In these applications, there may be hundreds of millions of spatial objects (e.g., shops, restaurants, pokémon, etc.) with a variety of distributions to be managed. Meanwhile, there may be huge numbers of service requests from users, e.g., Google Maps has reached one billion users [49], and Pokémon GO is attracting over 20 million daily active users [20]. Reporting POIs or pokémon in a given area in real time under such settings poses significant challenges.

Spatial indices are important techniques to address such challenges. They offer fast retrieval of spatial objects. We revisit a classic spatial index – the *R-tree* [27]. We aim to achieve an R-tree structure that is efficient in both window query processing and tree bulk-loading, and over both practical and worst-case workloads. R-trees have attracted extensive research interests [4, 7, 11, 28, 33, 51, 55] and have been implemented in industrial database systems [42, 43]. An R-tree is a balanced tree structure for external memory based spatial object indexing. Every node in an R-tree may contain multiple entries. In the leaf nodes, the entries are *minimum bounding rectangles* (MBR) of the data objects (and pointers to them); in the inner nodes, the entries are MBRs of and pointers to the child nodes. An R-tree node usually corresponds to a disk block, the size of which constrains the node *capacity*, i.e., the maximum number of entries per node, denoted by B. Given an R-tree, a window query returns all the data objects (e.g., POIs or pokémon) indexed in the tree that are within or intersect a given query window which is usually a rectangular region of interest.

R-trees have good query efficiency in practice when they are constructed with carefully crafted heuristics [11, 33, 36, 55]. However, all these heuristics cannot produce an R-tree with attractive performance guarantees in the worst case. The *Priority R-tree* (PR-tree) [7] is an R-tree with a non-trivial theoretical query performance guarantee. It answers a window query with $O((n/B)^{1-1/d} +$

k/B) I/Os in the worst case, which is known to be asymptotically optimal [4]. Here, n, d, and k denote the data set size, the dimensionality, and the output size, i.e., the number of objects satisfying the query, respectively. The PR-tree is designed for rectangles. As a follow-up study shows [28], the PR-tree may not have satisfactory empirical performance on data objects of a small size (e.g., point data) or queries with small query windows; and the tree construction is difficult to parallelize.

We re-examine the construction of R-trees and aim for high window query efficiency over point data, which is a common way for representing locations on digital maps. Spatial objects with extents can also be efficiently transformed into points for query processing [65]. We target application scenarios such as digital mapping where queries are much more frequent than updates over the data. We construct R-trees that are query cost optimal. Our R-trees can also handle dynamic data updates efficiently and without compromising the query cost optimality.

We propose an R-tree packing strategy that creates R-trees with the worst-case optimal window query I/O cost $O((n/B)^{1-1/d} + k/B)$. This strategy has a simple procedure and yields R-trees that have high practical query efficiency. A key step we take before packing the data points is to map them into a *rank space* such that their coordinates are mapped to their ranks in each dimension. Ties in one dimension are broken by the coordinates in the other dimension(s). As a result, we obtain data points with no repetitive coordinates in any dimension. We then simply pack every B data points into a leaf node (except possibly the last leaf node) of an R-tree in ascending order of the *Z-order* values of the data points in the rank space. The Z-order is an ordering created by the *Z-curve* [44] which is a common type of *space-filling curve* (SFC). The inner nodes of the R-tree are created by packing every B child nodes into a parent node (except possibly the last node in each level) again in ascending order of the Z-curve values and recursively from the bottom to the top of the tree. An inner node entry stores a pointer to a child node and its MBR.

Our R-tree packing strategy relies only on sorting. It takes $O(sort(n)) = O((n/B)\log_{M/B}(n/B))$ I/Os to bulk-load an R-tree, where M is the size of the memory. A key advantage of this strategy is that it is highly parallelizable, which is an important feature in the big data era. Bulk-loading an R-tree with this strategy well suits the popular *massively parallel communication* (MPC) model [5, 6, 10], which paves the foundation for designing algorithms for MapReduce systems [18]. We propose a parallel bulk-loading algorithm that takes $O(\log_s n)$ rounds of computation, where $s = n/g$ and g is the number of machines participating in the parallel algorithm. The (parallel) running time of our algorithm is $O((n \log n)/g)$, while the total time (summed over all machines) is $O(n \log n)$. The load of our algorithm, i.e., the maximum number of words received by any participating machine, is $O((n \log_s n)/g)$. For modern machines, s is large, e.g., at the order of millions, allowing us to bulk-load an R-tree with a very large number of points in just a few rounds of computation.

We further consider how to handle data updates for a bulk-loaded R-tree without impacting the worst-case query cost optimality. We first convert the bulk-loaded R-tree into a *deletion-only* structure by indexing the Z-order values of the data points (in addition to their MBRs) in the tree. This structure can answer window queries using the MBRs and handle data deletions using the Z-order values (just like a B-tree). It retains the $O(n/B)$ space cost and the $O((n/B)^{1-1/d} + k/B)$ window query I/O cost, while it can also handle a deletion in $O(\log_B n)$ amortized I/Os.

We then extend this structure to support insertions via the *logarithmic method* [13, 45]. The logarithmic method replaces dynamic data insertions with bulk-loading a series of up to $\lceil \log_B n \rceil$ R-trees, where the i-th R-tree holds at most B^i new data points. When a window query is issued, the query is run on every bulk-loaded R-tree. The worst-case query cost optimality of any individual R-tree is retained, since there is no dynamic insertions on these trees. The overall window query I/O cost does not exceed $O((n/B)^{1-1/d} + k/B)$ either. This is because the tree sizes form a geometric

series, the maximum of which is n. The overall query cost is dominated by that of the largest tree, which is $O((n/B)^{1-1/d} + k/B)$.

When applying the logarithmic method, we use a B-tree to record the id of the tree in which a data point is indexed. This B-tree helps identify the tree from which a data point is to be deleted. The tree ids in this B-tree may need to be updated when there are data insertions, which adds additional insertion costs. To reduce the insertion costs, we further modify our deletion-only structure by adding pointers that point from data points to the B-tree nodes. Such pointers enable efficiently locating the B-tree nodes to be updated for data insertions. As a result, compared with an earlier study on dynamization of bulk-loaded structures [8], we reduce the amortized insertion I/O cost from $O(\log_B^2 n)$ to $O(\log_B n)$ when $O(\log_{M/B}(n/B)) = O(1)$, i.e., the memory size M is at the scale of the data set size n, which is typically satisfied by modern machines.

While the rank space has been used by the computational geometry community to develop theoretical bounds [17, 21], we observe for the first time that rank-space conversion can be leveraged to build a worst-case optimal structure for window queries. Furthermore, it is perhaps surprising that we are able to achieve the purpose by combining the rank space with an SFC, because SFC-based indices were previously thought to have poor worst-case query costs. Indeed, as shown by Arge et al. [7], if an SFC is used directly (i.e., in the original data space) for indexing, there exist window queries which retrieve few points, but have I/O costs linear to the data set size. In fact, even analyzing the query cost of an SFC-based index is non-trivial. The limited literature on this topic [31, 39, 60] has focused on the average query cost, which is analyzed indirectly by studying the clustering behavior of SFCs.

In summary, this paper makes the following contributions:

(1) We propose the first SFC-based packing strategy that creates R-trees with a worst-case optimal window query I/O cost.
(2) The proposed packing strategy suggests a simple R-tree bulk-loading algorithm relying only on sorting. We propose such an algorithm under the massively parallel communication model (and thus, it works on MapReduce systems) with attractive performance guarantees.
(3) We propose R-tree based dynamic index structures to handle data updates. We show that such dynamic structures retain the optimal window query I/O cost in the worst-case. Further, compared with an earlier study on dynamization of bulk-loaded structures [8], we reduce the amortized data insertion cost from $O(\log_B^2 n)$ to $O(\log_B n)$.
(4) We perform extensive experiments on both real and synthetic data. The results confirm the superiority of the proposed R-tree packing strategy: on real data, the query I/O cost of the R-trees that we construct is up to 31% lower than that of PR-trees [7] and similar to that of STR-trees [36], which are a classic type of sorting based bulk-loaded R-trees; on highly skewed synthetic data, the query I/O cost of the R-trees that we construct is 54% lower than that of PR-trees and 64% lower than that of STR-trees. The proposed bulk-loading algorithm also outperforms the PR-tree bulk-loading algorithm in running time by 85% over large data sets with 20 million data points. When processing updates with the proposed dynamic index structures, we achieve up to 98% lower query I/O costs comparing with the LR-tree [16] – a Hilbert R-tree [33] variant with update supports. The advantage is most significant when the data distribution is highly skewed.

This article is an extension of our previous conference paper [50]. In the previous work, we presented the R-tree packing strategy based on SFCs in the rank space. We showed the worst-case query I/O cost optimality and the parallel implementation of the strategy. In this article, we present new techniques to handle data updates to the R-trees constructed by our packing strategy while retaining the worst-case query I/O cost optimality (Section 5). As a result, we obtain a fully dynamic

index structure that is worst-case optimal and empirically efficient for window query processing. Further, we show that our techniques can reduce the amortized data insertion cost from $O(\log_B^2 n)$ to $O(\log_B n)$, comparing with an earlier study on dynamization of bulk-loaded structures [8]. Our additional experiments on the index update techniques show (i) the effectiveness of the techniques for retaining the high query efficiency of our index structure, and (ii) the efficiency of the techniques in handling updates to our index structure (Section 6.2.3). We have also added a literature review on R-tree update techniques (Section 2).

The rest of the article is organized as follows: Section 2 reviews related work. Section 3 details the proposed R-tree packing strategy and the worst-case window query I/O costs. Section 4 describes the proposed parallel R-tree bulk-loading algorithm. Section 5 discusses data update handling. Section 6 presents experimental results, and Section 7 concludes the paper.

2 RELATED WORK

We review studies on spatial queries and access methods, with a focus on R-trees.

Spatial queries and access methods. We focus on the window query (rectangular range query) which is a basic type of spatial query [26]. A window query returns all data objects that satisfy a certain predicate with a given query window, i.e., a (hyper)rectangular region of interest. Common query predicates include containment and intersection, which require the data objects to be fully contained in or intersect the query window, respectively.

A straightforward window query algorithm sequentially checks every data object and returns an object if it satisfies the query predicate. This algorithm takes $O(n/B)$ I/Os regardless of data distribution and output size. Spatial indices have been used to obtain higher query efficiency. We focus on the R-tree index [27]. For a comprehensive review on spatial indices and spatial query processing, interested readers are referred to [22].

R-trees. As discussed earlier, the R-tree is a balanced tree structure. The *maximum* number of entries per tree node (node capacity) B is constrained by the disk block size, while the *minimum* number of entries per tree node (except the root node) is $\Omega(B)$. The root node needs to contain at least two entries unless it is also a leaf node. Thus, the height of an R-tree indexing n objects is bounded by $O(\log_B n)$.

A window query is processed by a top-down traversal over the nodes of an R-tree whose MBRs satisfy the query. When the leaf nodes are reached, data objects in them satisfying the query are returned. A series of studies [11, 14, 30, 41, 55] propose heuristics to optimize the node MBRs during dynamic data insertion. The *R*-tree* [11], for example, considers the MBR overlaps and region perimeters to decide the node into which a new object should be inserted.

R-tree packing and bulk-loading. A different stream of research considers how to construct an R-tree by packing data objects into the leaf nodes directly rather than inserting them individually. The entire R-tree is bulk-loaded in a bottom-up fashion. Most R-tree packing algorithms [1, 19, 28, 33, 36, 52] rely on some ordering of the data objects and hence have an I/O cost of $O((n/B)\log_{M/B}(n/B))$, which is the cost for sorting n objects (recall that M is the number of objects allowed in the main memory). Specifically, Roussopoulos and Leifker [52] sort the data objects by their x-coordinates and pack every B objects into a leaf node. Leutenegger et al. [36] first sort the data objects by their x-coordinates, and then partition the data into $\sqrt{n/B}$ subsets. Objects in each subset are sorted by their y-coordinates and packed into the leaf nodes. Other studies use the *Hilbert ordering* [19, 28, 33]. Their resultant R-trees have shown good window query performance on nicely distributed data [28]. Achakeev et al. [1] also use an SFC (e.g., a Hilbert curve) for object ordering. Instead of packing every B objects into a leaf node, they compute a series of split points to split the list of sorted objects

and pack the objects into leaf nodes accordingly. Their aim is to minimize the sum of the MBR areas of the resultant tree nodes. These R-trees are not worst-case optimal for window queries.

There are also top-down bulk-loading algorithms. The *Top-down Greedy Split* (TGS) algorithm [23] is a typical example. TGS partitions the data set into two subsets repeatedly until B approximately equisized subsets have been obtained. The MBRs of these B subsets form entries of the root. Each partition uses a cut orthogonal to an axis that yields two subsets with the minimum sum of costs, where the cost is based on a user-defined function, e.g., the area of the MBR of a subset. There are $O(B)$ candidate cuts, where the hidden constant lies in the different cuts in different dimensions and on different orderings (e.g., lower x corner, center, etc.). In each dimension and with a particular ordering, the i-th cut puts $i \cdot (n/B)$ objects in one subset and the rest in the other subset. TGS has been shown to produce R-trees with good query efficiency, but it has a high worst-case I/O cost, $O(n \log_B n)$, for R-tree construction. This is because it needs to scan the data set B times to create the B partitions of a node (assuming that the orderings used for partitioning have been precomputed). If viewed from a recursive binary partition perspective, the I/O cost of TGS is effectively $O((n/B) \log_2 n)$ [7].

Agarwal et al. [4] propose an algorithm to bulk-load a *Box-tree*, which can be converted to an R-tree with a worst-case query I/O cost of $O((n/B)^{1-1/d} + k \log_B n)$. This work is more of theoretical interest. No implementation or experimental results have been given for the algorithm.

The *PR-tree* [7] is an R-tree that offers a worst-case window query I/O cost of $O((n/B)^{1-1/d} + k/B)$, which is asymptotically optimal [4]. A PR-tree is created from a *pseudo-PR-tree*, which is an unbalanced tree built in a top-down fashion. To create a pseudo-PR-tree, the data set is partitioned into six partitions to form the child nodes of the root. Four of the partitions contain B objects each, which are objects with the smallest lower x-coordinates, the smallest lower y-coordinates, the largest upper x-coordinates, and the largest upper y-coordinates, respectively. The remaining two partitions are two equisized partitions of the remaining objects, which are then recursively partitioned to form subtrees of the root. When a pseudo-PR-tree is created, its leaf nodes are used as the leaf nodes of a PR-tree. The MBRs of the leaf nodes are used to create another pseudo-PR-tree, the leaf nodes of which are used as the parent nodes of the leaf nodes of the PR-tree. A PR-tree is then built with $O((n/B) \log n)$ I/Os bottom-up. Arge et al. [7] further propose a bulk-loading strategy that lowers the I/O cost to $O((n/B) \log_{M/B}(n/B))$. The main issue of the PR-tree is that it lacks practical efficiency in answering queries with small query windows or over data objects of a small size (e.g., point data) [28].

We also note that other spatial indices such as *kd-trees* [12], *O-trees* [34], and *cross-trees* [25] can offer a worst-case optimal query I/O performance. Compared with R-trees created by our packing strategy, kd-trees are more difficult to bulk-load in parallel. In the MPC model, Agarwal et al. [5] propose a randomized algorithm that can bulk-load a kd-tree with $O(\text{polylog}_s\, n)$ rounds of computation. In contrast, we can bulk-load an R-tree with $O(\log_s n)$ rounds of computation which is lower, and our bulk-loading algorithm is deterministic. As for O-trees, they do not belong to the R-tree family. They combine multiple auxiliary structures to ensure their theoretical guarantees. This approach is mainly of theoretical interest, but in practice is expensive in both space consumption and query cost (even being asymptotically optimal in the worst case). Cross-trees share a similar issue in its practical query performance [3]. These indices are not discussed further.

R-tree update handling. The dynamic R-tree construction heuristics [11, 14, 30, 41, 55] mentioned above (e.g., the R*-tree insertion heuristics [11]) can also handle R-tree updates. Such heuristics, however, do not guarantee optimal query performance for the updated R-trees.

Studies on R-tree packing and bulk-loading focus on static data settings. Most of them either do not consider updates at all [4, 23, 36] or simply use the above dynamic R-tree construction heuristics

for updates [52]. The Hilbert R-tree [28] takes a slightly different update handling strategy. This R-tree can be seen as a B-tree that indexes the Hilbert-order values of the data points, and its updates are handled by B-tree update algorithms. None of these studies guarantee optimal query performance, with or without updates.

Arge et al. [7] extend the PR-tree to an *LPR-tree* using the logarithmic method [13, 45] for handling updates while retaining the worst-case optimal window query performance. The LPR-tree consists of a series of *annotated pseudo-PR-trees* (APR-trees) with increasing sizes. An APR-tree is a pseudo-PR-tree with additional aggregate information stored in the inner nodes of the tree. Data insertions on the LPR-tree are handled by bulk-loading new (and larger) APR-trees over the data points to be inserted together with the data points in the existing (and smaller) APR-trees. Data deletions, on the other hand, are handled by deleting the data points directly from the APR-trees that contain them. To help locate a data point to be deleted, a *time index* is used to keep track of the APR-tree in which a data point is contained. To further improve the update efficiency, an $O(M)$ sized component of the LPR-tree is kept in main memory, which includes the first $\log_2(M/B)$ APR-trees and the top levels of the rest of the APR-trees. By doing so, the LPR-tree obtains an amortized insertion I/O cost of $O(\log_B(n/M) + (1/B)\log_{M/B}(n/B) \cdot \log_2(n/M))$ and an amortized deletion I/O cost of $O(\log_B(n/M))$. The LPR-tree is more of theoretical interest. No implementation or experimental results have been given for it. Our proposed dynamic structure also uses the logarithmic method, but it differs from the LPR-tree in that it is built on R-trees directly which are much easier to construct and update than the APR-trees. This enables us to implement our dynamic structure and evaluate its empirical performance. Also, our dynamic structure does not need to reside in main memory which is more flexible. To the best of our knowledge, our dynamic structure is the first dynamic structure that is built on the R-trees directly while retaining the worst-case optimal window query performance.

Bozanis et al. [16] apply the logarithmic method over the Hilbert R-tree and propose the *LR-tree*. The LR-tree bulk-loads new Hilbert R-trees to handle data insertions. It uses a simplified R*-tree deletion algorithm without node merging to handle data deletions. Since the underlying R-trees in the LR-tree are not window query optimal, the LR-tree does not guarantee worst-case optimal window query performance either. Regardless of this, the LR-tree is the closest structure to our proposed dynamic structure. Thus, it is used as a baseline in our experiments.

Parallel R-tree management. Parallelism has been exploited to scale R-trees to large data sets and user groups. An early study [32] considers storing an R-tree on a multi-disk system. It stores a newly created tree node in the disk that contains the most dissimilar nodes to optimize the system throughput. A few studies [35, 38, 54] assume a shared-nothing (client-server) architecture for distributed R-tree storing and query processing. Koudas et al. [35] store the inner nodes on a server while the leaf nodes are stored on clients. They study how to decide the number of data objects to be stored on a client and which objects to be stored together. Schnitzer and Leutenegger [54] further create local R-trees on clients for higher query efficiency. Mondal et al. [38] study load balancing for R-trees in shared-nothing systems.

The studies above do not focus on parallel R-tree bulk-loading. Papadopoulos and Manolopoulos [48] propose a generic procedure for parallel spatial index bulk-loading. They use sampling to estimate the data distribution which helps partition the data space into regions. Data objects in different regions are assigned to different clients for local index building. A global index is built on the server to serve as a coordinator for query processing. A more recent study [2] bulk-loads an R-tree with the MapReduce framework level by level, where each level takes a MapReduce round. It uses the bulk-loading strategy mentioned above that aims to minimize the sum of the MBR areas of the tree nodes [1]. Similar ideas have been used on GPUs [61] without a cost analysis.

Table 1. Frequently Used Symbols

Symbol	Description
P	A data set
p	A data point
n	The cardinality of P
d	The dimensionality of P
q	A window query
k	The answer set size of a window query
T	An R-tree
B	The node capacity of an R-tree
h	The height of an R-tree
M	The memory size of a standalone machine
g	The number of machines in a cluster
s	The number of data points allowed in a machine
Γ	An R-tree (B-tree) indexing both MBRs and Z-order values of data points
Λ_{id}	A B-tree mapping data point ids to Z-order values and tree ids
H	The number of sub-trees in a LogR*-tree
μ	The number of data point updates processed
η	The number of global rebuilds
$\widetilde{n}_j$	The number of data point at the j-th global rebuild

3 R-TREE PACKING

We consider a set P of n data points in a d-dimensional Euclidean space. For ease of presentation, we use $d = 2$ in our examples, although our approach applies to an arbitrary fixed dimensionality $d \geq 2$. We focus on window queries. Given a rectangle q, a window query reports all the points in $P \cap q$. We list the frequently used symbols in Table 1.

3.1 Mapping to Rank Space

Before creating an index structure over P, we first map the data points into a d-dimensional *rank space* as follows. In each dimension of the original data space, we sort the data points by their coordinates and use the ranks as the coordinates in the corresponding dimension of the rank space. If two data points have the same rank in a dimension, we break the tie by further comparing their coordinates in the other dimensions (of the original space) in the order of dimension 1, dimension 2, . . ., dimension d. We assume no data points with the same coordinates in all d dimensions.

Define by $[n]$ the integer domain $[0, n-1]$. After the mapping, *P becomes a set of n d-dimensional points in $[n]^d$ such that no two points share the same coordinate in any dimension.*

We use Fig. 2 to illustrate the mapping with a set of 8 ($n = 8$) 2-dimensional points $P = \{p_1, p_2, \ldots, p_8\}$. The coordinates of the points in the rank space are their ranks in the original space. For example, p_1 has the smallest x-coordinate and second largest y-coordinate in the original space. Thus, its x-coordinate and y-coordinate in the rank space are 0 and 6, respectively. Points p_2 and p_3 both have the second smallest x-coordinate in the original space. In the rank space, their x-coordinates are 1 and 2, because p_2 has a smaller y-coordinate in the original space.

A query rectangle $q = [ql_{e1}, qh_{e1}] \times [ql_{e2}, qh_{e2}] \times \ldots \times [ql_{ed}, qh_{ed}]$ in the original space is mapped to a rectangle $q = [ql_1, qh_1] \times [ql_2, qh_2] \times \ldots \times [ql_d, qh_d]$ in $[n]^d$. Here, ql_i is the smallest rank of the data points whose coordinates in dimension i in the original space are greater than or equal

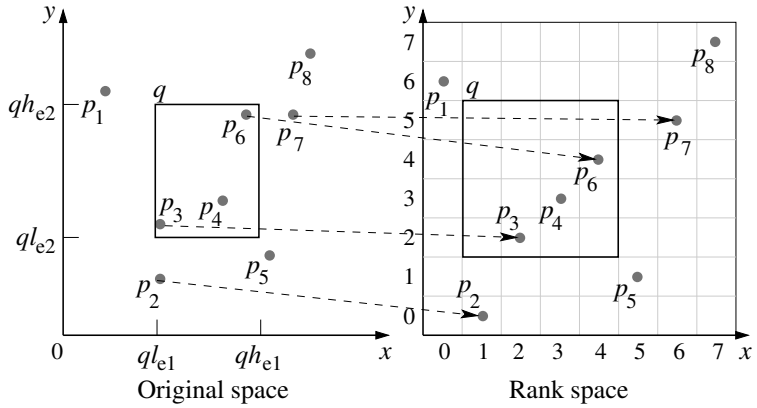


Fig. 2. Mapping to rank space

to ql_{e1}; qh_i is the largest rank of the data points whose coordinates in dimension i in the original space are smaller than or equal to qh_i. In Fig. 2, the solid rectangle represents a query rectangle q. In the original space, the query range $[ql_{e1}, qh_{e1}]$ spans p_2, p_3, p_4, and p_6 in the x-dimension, among which p_2 (p_6) has the smallest (largest) rank 1 (4). Thus, $[ql_{e1}, qh_{e1}]$ is mapped to $[1, 4]$ in the rank space. Similarly, the query range $[ql_{e2}, qh_{e2}]$ is mapped to $[2, 5]$ in the rank space. Note that the query mapping does not introduce false positives in the query answer because the data points do not share the same coordinate in either dimension in the rank space.

Our goal is to store P in a structure so that all window queries can be answered efficiently in the worst case. Without loss of generality, we consider that n is a power of 2. Our techniques work straightforwardly for the case where n is not a power of 2, which is discussed in Section 3.3.1.

3.2 Tree Structure and Packing Strategy

Our structure is simply an R-tree where the leaf nodes are obtained by packing points in ascending order of their *Z-order* [44] values. Other space-filling curves such as *Hilbert curves* can also be used (as will be discussed in Section 3.4) but Z-order is used for illustration.

For each point $p \in P$, we compute its Z-order value $Z(p)$ in $[n]^d$ by interleaving the bits of its coordinate in every dimension. For example, suppose $p = (x, y)$, where $x = \alpha_1\alpha_2 \ldots \alpha_l$ and $y = \beta_1\beta_2 \ldots \beta_l$ in binary form where $l = \log_2 n$. Then, $Z(p) = \beta_1\alpha_1\beta_2\alpha_2 \ldots \beta_l\alpha_l$. We sort all the points of P by their Z-order values, and cut the sorted list into subsequences, each of which has length B, except possibly the last subsequence. Here, $B \geq 1$ is a parameter that controls the maximum number of points that fit in a leaf node of an R-tree. Each leaf node includes the points in a subsequence. The inner nodes of the R-tree are created by packing every B child nodes into a parent node (except possibly the last node in each level) recursively from the bottom to the top of the tree. This process resembles how a B-tree is created, except that an inner node entry stores a pointer to a child node and its MBR instead of a key value. This creates our target R-tree.

The R-tree packing strategy is illustrated in Fig. 3. The rank space can be seen as an 8×8 grid. A *Z-curve* (the dotted polyline) is drawn across the rank space. The order that a cell is reached by the curve is the Z-order value of the data point in the cell, e.g., in Fig. 3a, p_2 is in the second cell reached by the curve; its Z-order value is 1, which is labeled in parentheses next to p_2 (same for the other points). Based on the Z-order values, the data points are sorted as: $\langle p_2, p_3, p_4, p_5, p_1, p_6, p_7, p_8 \rangle$. We use $B = 2$ in this example. The eight data points are packed into four leaf nodes: $N_1 = \langle p_2, p_3 \rangle$, $N_2 = \langle p_4, p_5 \rangle$, $N_3 = \langle p_1, p_6 \rangle$, and $N_4 = \langle p_7, p_8 \rangle$. These four leaf nodes are then packed in the order of the Z-order values of the data points stored in them, resulting in two inner nodes $N_5 = \langle N_1, N_2 \rangle$

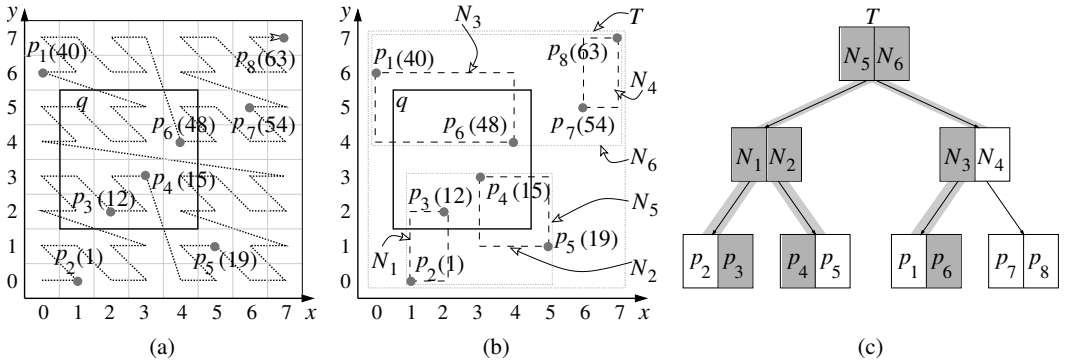


Fig. 3. R-tree packing

and $N_6 = \langle N_3, N_4 \rangle$. A root node is further created to point to N_5 and N_6. The MBRs and the R-tree T created are shown in Figs. 3b and 3c.

Algorithm 1 summarizes the the proposed R-tree packing strategy with the help of an auxiliary queue Q. The queue stores 2-tuples in the form of $\langle N, t \rangle$ where N and t are a tree node and its level in the tree, respectively. The packing strategy takes:

(1) a sort on the data points for each dimension to map them into the rank space (Line 1),
(2) a linear scan on the data points to compute their Z-order values (Lines 2 and 3),
(3) another sort on the Z-order values (Line 4), and
(4) another linear scan on the data points (Lines 6 to 8) and $\log_B n - 1$ rounds of linear scans on the MBRs of tree nodes for packing and loading an R-tree (Lines 9 to 12).

Together, this packing strategy takes $O((n/B)\log_{M/B}(n/B))$ I/Os to bulk-load an R-tree (the CPU time is $O(n \log n)$, noticing that the Z-order value of a point can be calculated in $O(\log n)$ time). Sorting and linear scans can be easily parallelized. This suggests a simple parallel R-tree bulk-loading algorithm where everything boils down to sorting. We present such an algorithm in Section 4.

Algorithm 1: Build-R-tree

Input: $P = \{p_1, p_2, \ldots, p_n\}$: a d-dimensional database; B: the capacity of a tree node.
Output: T: an R-tree over P.

1 Map P into the rank space;
2 **for** *each* $p_i \in P$ *in the rank space* **do**
3 Compute Z-order value of p_i;
4 Sort P in ascending order of the Z-order values;
5 Let $Q \leftarrow \emptyset$;
6 **for** *every* B *data points in the sorted* P **do**
7 Create a leaf node N to store the B data points;
8 $Q.enqueue(\langle N, 1 \rangle)$;
9 **while** $Q.size() > 1$ **do**
10 Dequeue the first B nodes of the same level t from Q;
11 Create a node N to store MBRs (pointers) of the nodes;
12 $Q.enqueue(\langle N, t + 1 \rangle)$;
13 Let T point to the last node in Q;
14 **return** T;

3.3 Window Query Processing

When a window query is issued, we first map it to the rank space following the procedure described in Section 3.1. To facilitate fast mapping, we create a B-tree for each dimension to store pairs of point coordinates in the original space and corresponding coordinates in the rank space. Query mapping using B-trees takes $O(\log_B n)$ I/Os. The mapped query is then answered by our R-tree in the same way as for a conventional R-tree. We omit the pseudo-code of the query algorithm for conciseness. As an example, in Fig. 3c, we show the search paths for processing query q in gray.

3.3.1 Query Cost. We prove that our R-tree answers a window query with $O((n/B)^{1-1/d} + k/B)$ I/Os in the worst case, where k is the number of points reported. This query complexity is known to be asymptotically optimal [4, 15]. We start with the case where $d = 2$ and prove the worst-case query cost to be $O(\sqrt{n/B} + k/B))$ in this subsection. We will generalize the proof to an arbitrary fixed dimensionality $d \geq 2$ in the next subsection.

Let $h \leq \log_B n$ be the height of the tree. Label the levels of the tree as $1, 2, \ldots, h$ bottom up. Consider any level $t \in [1, h]$. Let ℓ be any vertical line in $[n]^2$. We prove the following lemma, which is sufficient for establishing our claim.

Lemma 3.1. *The line ℓ intersects the MBRs of $O(\sqrt{n/B^t})$ nodes at level t.*

Proof. Intuitively, the MBR of a node intersects a line ℓ when it covers data points on both sides of ℓ (e.g., in Fig. 4, node N_3 contains p_1 and p_6 on both sides of the dashed line ℓ) or data points on ℓ (e.g., p_2 in N_1). Such a node corresponds to a Z-curve segment that crosses or ends at ℓ. Since different nodes correspond to non-overlapping curve segments (because the data points are packed by ascending Z-order values), we derive the number of nodes intersecting ℓ via the number of times that the Z-curve crosses ℓ.

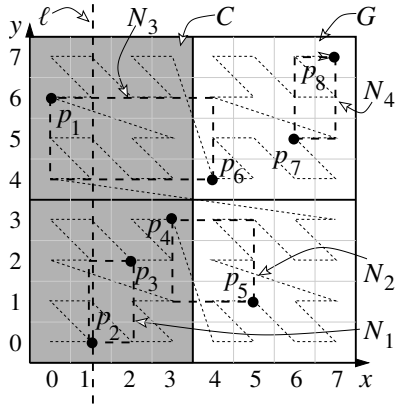


Fig. 4. Window query I/O cost

Let m be the smallest power of 2 larger than or equal to $\sqrt{nB^t}$. Divide $[n]^2$ into an $(n/m) \times (n/m)$ grid denoted by G, where each cell has m^2 locations in $[n]^2$. Note that the Z-curve traverses all the locations in a cell before moving to another, i.e., it never comes back to the same cell.

We use Fig. 4 to illustrate the proof for the case where $t = 1$, i.e., at the leaf node level. It shows the four leaf nodes N_1, N_2, N_3, and N_4 (the dashed rectangles) of an R-tree constructed. We have $\sqrt{nB^t} = \sqrt{8 \times 2^1} = 4$ which means $m = 4$. The rank space is divided into an $(8/4) \times (8/4) = 2 \times 2$ grid, as denoted by the black solid line grid. The Z-curve enters and leaves each cell once, e.g., for the top-left cell, the Z-curve enters at its bottom-left corner and leaves from its top-right corner.

Let $C = [a, b] \times [n]$ be the column of G that contains line ℓ. In Fig. 4, the vertical dashed line represents ℓ, which is in column $C = [0, 3] \times [8]$ highlighted in gray. Define the line $x = a$ as

the left boundary of C, and the line $x = b$ as the right boundary. Let u be a node whose MBR intersects ℓ. Define $X(u)$ as the x-range of the MBR of u. For example, node N_3 intersects the line, and $X(N_3) = [0, 4]$. Such a node u can be one of the following types:

- Type 1: $a \in X(u)$ or $b \in X(u)$, i.e., u overlaps column C (cf. node N_3).
- Type 2: $X(u) \subset [a, b]$, i.e., u is inside column C (cf. node N_1).

We prove that there are at most $2n/m \leq 2\sqrt{n/B^t}$ nodes of Type 1 and $O(1 + m/B^t) = O(\sqrt{n/B^t})$ nodes of Type 2, which completes the proof of the lemma.

- Type 1: Note that the Z-curve crosses the left boundary (i.e., enters column C) n/m times. This is because there are n/m cells of G in each column, and the curve enumerates all the locations of a cell of G before moving to another. In Fig. 4, there are $n/m = 8/4 = 2$ cells in column C. The curve enters these two cells once each. Since the data points are sorted and packed into nodes by their curve values, there are at most n/m nodes that contain data points on both sides of the left boundary. Otherwise, some of these nodes must have overlapping curve values, which is against our packing strategy. The same applies to the right boundary. Thus, there are at most $2n/m$ nodes of Type 1. In the figure, N_3 overlaps the right boundary of the top cell of column C. It contains two data points p_1 and p_6 on the two sides of the right boundary of this cell. The curve segment between them crosses the right boundary of the cell. Since the curve only leaves the cell once, there cannot be another node N that also overlaps the right boundary of the cell. Otherwise, the two curve segments corresponding to N and N_3 must overlap, which violates our packing strategy.
- Type 2: When u is in column C, the x-coordinate of any data point in the subtree of the node is in the range of $[a, b]$. There are $b - a + 1 = m$ distinct x-coordinates in the range, implying m data points in the range (recall that all points have distinct x-coordinates). Each node at level t can index B^t data points. Thus, there are $O(1 + m/B^t)$ nodes of Type 2. In Fig. 4, $b - a + 1 = 3 - 0 + 1 = 4$. The four data points in the gray column can form at most $m/B^t = 4/2^1 = 2$ nodes fully contained in the column, although there is just one such node in this example which is N_1. If p_5 does not exist then p_4 and p_1 will form another node fully contained in the column.

□

Discussion. For an R-tree with a height $h \leq \log_B n$, line ℓ interests the MBRs of $O(\sqrt{n/B}) + O(\sqrt{n/B^2}) + \ldots + O(\sqrt{n/B^h}) = O(\sqrt{n/B})$ nodes. The above proof assumed n being a power of 2 and $d = 2$. When n is not a power of 2, let $\rho = \lceil \log_2 n \rceil$. We enlarge the rank space to $[2^\rho]^2$. Line ℓ intersects $O(\sqrt{2^\rho/B})$ nodes in the enlarged rank space. We have $O(\sqrt{2^\rho/B}) = O(\sqrt{n/B})$ because $2^\rho \leq 2n$. The above argument can also be generalized to an arbitrary fixed dimensionality $d \geq 2$ to prove that our query cost is bounded by $O((n/B)^{1-1/d} + k/B)$ in the worst case. This will be proven in the following Section 3.3.2, which proves an even stronger result that *subsumes* the aforementioned bound as a special case.

3.3.2 Query Sensitive Bound in Arbitrary Dimensionality. Consider a query rectangle $q = [ql_1, qh_1] \times [ql_2, qh_2] \times \ldots \times [ql_d, qh_d]$ in $[n]^d$, where $d \geq 2$ is an arbitrary fixed dimensionality. For each $i \in [1, d]$, set $\delta_i = qh_i - ql_i + 1$, and let $Z_i = \{1, 2, \ldots, d\} \setminus \{i\}$, namely, Z_i includes all the integers from 1 to d except i. We prove a stronger version of our previous lemma: our structure answers the query in

$$O(\log_B n + \Delta^{1/d}/B^{1-1/d} + k/B) \tag{1}$$

I/Os where $\Delta = \sum_{i=1}^{d} \prod_{j \in Z_i} \delta_j$. In this bound, the three components $\log_B n$, $\Delta^{1/d}/B^{1-1/d}$, and k/B denote the costs to map the query to rank space (query q here is already mapped), to find the nodes intersecting the query boundary, and to output the points within the query, respectively.

The bound looks a bit unusual such that it would help to look at some special cases: for $d = 2$, the query cost is $O(\log_B n + \sqrt{(\delta_1 + \delta_2)/B} + k/B)$, while for $d = 3$, the cost becomes $O(\log_B n + (\delta_1\delta_2 + \delta_1\delta_3 + \delta_2\delta_3)^{1/3}/B^{2/3} + k/B)$. Since $\delta_i \leq n$ for all $i \in [1, n]$, it always holds that $\prod_{j \in Z_i} \delta_j \leq n^{d-1}$ and $\Delta \leq d \cdot n^{d-1}$. Thus, Equation 1 is bounded by $O((d \cdot n^{d-1})^{1/d}/B^{1-1/d} + k/B) = O((n/B)^{1-1/d} + k/B)$. In other words, Equation 1 is never worse than the (query *insensitive*) bound established in Section 3.3.1, but could be substantially better when q is small.

We say that the MBR of a node *partially intersects* q if it has a non-empty intersection with q, but is not contained by q. We prove the following lemma, which is sufficient for establishing our claim.

Lemma 3.2. *The query rectangle q partially intersects the MBRs of $O(1 + \Delta^{1/d}/(B^t)^{1-1/d})$ nodes at level t.*

Proof. Let m be the smallest power of 2 at least $(\Delta \cdot B^t)^{1/d}$. Divide $[n]^d$ into a grid G of size:

$$\underbrace{(n/m) \times (n/m) \times \ldots \times (n/m)}_{d}$$

Each cell of G has m^d locations in $[n]^d$ (the cell's projection on each dimension covers m values). For each $i \in [1, d]$, define a *dimension-i column* of G as the maximal set of cells in G that have the same projection on dimension i. Grid G has n/m dimension-i columns, each of which is a d-dimensional rectangle in $[n]^d$ that covers the entire range $[n]$ on every dimension $j \neq i$.

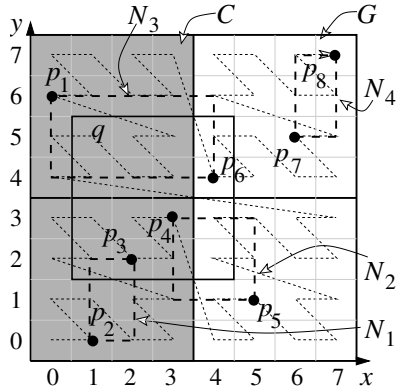


Fig. 5. Query sensitive I/O cost

We use Fig. 5 to illustrate the proof, where $d = 2$ and $t = 1$, i.e., we consider the leaf node level. We have $q = [1, 4] \times [2, 5]$ (the solid line rectangle) and hence $\delta_1 = 4 - 1 + 1 = 4$ and $\delta_2 = 5 - 2 + 1 = 4$. Thus, $m = (\Delta \cdot B^t)^{1/d} = \sqrt{(\delta_1 + \delta_2)B^t} = \sqrt{(4 + 4) \times 2^1} = 4$. The rank space is divided into an $(8/4) \times (8/4) = 2 \times 2$ grid, which is represented by the black solid line grid. Grid G has $8/4 = 2$ dimension-1 (the x-dimension) columns, i.e., the two vertical columns.

A node whose MBR partially intersects q must intersect one of the $2d$ boundary faces of q (e.g., edges of q in Fig. 5). We will prove that there can be at most $O(1 + m/B^t + \prod_{j \in Z_i} \delta_j/m^{d-1})$ nodes intersecting each of the two faces of q perpendicular to dimension i. Summing this up on all d dimensions gives the desired upper bound on the total number of nodes that partially intersect q:

$$\begin{aligned} &\textstyle\sum_{i=1}^{d} O(1 + m/B^t + \prod_{j \in Z_i} \delta_j/m^{d-1}) \\ &= O(d + dm/B^t + \Delta/m^{d-1}) \\ &= O(1 + \Delta^{1/d}/(B^t)^{1-1/d}) \end{aligned}$$

Due to symmetry, it suffices to consider the face of q that corresponds to ql_1 (i.e., perpendicular to dimension 1) – we refer to it as the *dimension-1 left face* of q. Let C be the dimension-1 column

of G that covers this face; C is a rectangle that can be written as $[a, b] \times \underbrace{[n] \times [n] \times \ldots \times [n]}_{d-1}$ for some a, b satisfying $b - a + 1 = m$ and b is a multiple of 2. In Fig. 5, dimension 1 is the x-dimension, and C is the gray column $[0, 3] \times [8]$. Define the *left boundary* (or *right boundary*) of C to be the set of points in $[n]^d$ with coordinate a (or b, respectively) on dimension 1.

Let u be a level-t node with an MBR intersecting the dimension-1 left face of q, and $X(u)$ be the projection of the MBR of u on dimension 1, e.g., node N_3 intersects the left edge of q, and $X(N_3) = [0, 4]$. Such a node u can be one of the following types:

- Type 1: $a \in X(u)$ or $b \in X(u)$.
- Type 2: $X(u) \subset [a, b]$.

Next, we analyze the number of nodes for each type.

- Type 1: The Z-curve crosses the left boundary of C at most $O(\prod_{j=2}^{d} \lceil \delta_i/m \rceil) = O(1+\delta_2\delta_3 \ldots \delta_d / m^{d-1})$ times within the dimension-1 left face of q. This is because there are $O(\prod_{j=2}^{d} \lceil \delta_i/m \rceil)$ cells of C within the range of q in dimensions 2 to d (e.g., in Fig. 5, there are $1 + \delta_2/m = 1 + 4/4 = 2$ cells of C within the dimension 2 range $[2, 5]$ of q), and the curve enumerates all the locations of a cell before moving to another cell. By the reasoning explained in the proof of Lemma 3.1, there are at most $O(\prod_{j=2}^{d} \lceil \delta_i/m \rceil)$ nodes containing data points on both sides of the left boundary. The same applies to the right boundary. Therefore, the number of Type-1 nodes is $O(1 + \delta_2\delta_3 \ldots \delta_d/m^{d-1})$.
- Type 2: All the B^t points in the subtree of node u must have x-coordinates between a and b. There can be only $b - a + 1 = m$ such points (recall that all points have distinct coordinates in each dimension), implying at most $O(1 + m/B^t)$ such nodes.

It thus follows that the dimension-1 left face of q intersects the MBRs of $O(1+m/B^t+\prod_{j\in Z_i} \delta_j/m^{d-1})$ nodes at level t. This completes the proof. □

3.4 Extending to Other Space-Filling Curves

Although we have used the Z-curve as the representative SFC, the only property that we require from the Z-curve is the following *quad-tree recursive pattern*. Divide the data space $[n]^d$ (where n is a power of 2) into 2^d rectangles of the same size, i.e., each rectangle is a "d-dimensional square" with a projection length of $n/2$ on each dimension (recall how the root of a d-dimensional quad-tree would partition the space). For example, in Fig. 5, grid G is divided into $2^2 = 4$ squares (cells) each with an edge length of 4. The quad-tree recursive pattern says that the SFC must first enumerate all the points within a rectangle before starting to enumerate the points of another. In Fig. 5, the Z-curve enumerates the points of the bottom-left cell before moving to the bottom-right cell. The pattern must be followed recursively within each rectangle, by treating it as a smaller data space $[n/2]^d$. All our proofs hold *verbatim* on any SFCs (e.g., the Hilbert curve) that obey this pattern.

4 PARALLEL R-TREE BULK-LOADING

Next, we present a parallel R-tree bulk-loading algorithm based on our packing strategy. A straightforward parallel algorithm that bulk-loads an R-tree level by level requires $O(\log_B n)$ rounds of parallel computation. We show how to reduce the number of rounds to $O(\log_s n)$ without sacrificing the computation time. Here, s denotes the number of data points that a machine participating in the parallel algorithm can handle. Modern machines can easily handle millions of data points, where $\log_s n$ is typically bounded by a constant.

The key idea of the proposed algorithm is to distribute the data points (or MBRs of tree nodes) in a way that the machines can bulk-load $O(\log_B s)$ levels of the final R-tree in each round of parallel

computation. Then, $O(\log_s n)$ such rounds suffice to build the entire R-tree of $\log_B s \cdot \log_s n = \log_B n$ levels. To bulk-load $\log_B s$ levels in each round, a machine is assigned a subset of the data points (MBRs) that forms a few R-tree branches of $\log_B s$ levels independently from the data assigned to the other machines. This is feasible because we can assign data points to the machines in their sorted order for packing independently.

4.1 Parallel Computation Model

Without relying on a particular parallel platform such as Apache Hadoop, we design the parallel bulk-loading algorithm based on a generalized parallel model named the *massively parallel communication* (MPC) model [5, 6, 10]. Popular parallel frameworks such as MapReduce [18] and Spark [62] are typical examples of this model. Our implementation differs slightly from that of Agarwal et al. [5] who also use the MPC model to build an index. We copy the built index back to a single machine, while Agarwal et al. leave the index distributed among the machines. This is because our query algorithm runs on a single machine. We leave distributed query processing for future work.

The MPC model makes the following assumptions. Let n be the input size, g be the number of machines, and $s = n/g$. In each round of parallel computation, every machine receives some data from other machines, performs computation, and sends some data to other machines. The computation is done in the memory and hence there is no disk I/O cost, except for the initial data reading and the final data writing. We consider only algorithms that require a machine to receive/send $O(s)$ words of information in each round, i.e., the communication I/O cost for a machine in each round is $O(s)$ (with the terminology of Beame et al. [10], these algorithms must have *load* $O(s)$ in each round).

MPC algorithms are measured by:

(1) the *number of computation rounds* $\mathcal{R}$,
(2) the (parallel) *running time* $\mathcal{T}$, which sums up the *maximum* computation cost of a *single machine* in each round, and
(3) the *total amount of computation* $\mathcal{W}$, which sums up the computation costs of *all machines* in all rounds.

Let $t_{\mathcal{M}_{i,r}}$ be the time complexity of machine $\mathcal{M}_i$ in round r. Then:

$$\mathcal{T} = \sum_{r=1}^{\mathcal{R}} \max_{i \in 1..g} t_{\mathcal{M}_{i,r}} \tag{2}$$

$$\mathcal{W} = \sum_{r=1}^{\mathcal{R}} \sum_{i=1}^{g} t_{\mathcal{M}_{i,r}} \tag{3}$$

For the purpose of building an R-tree, $\mathcal{W}$ should not exceed the time complexity $O(n \log n)$ for a single-machine implementation of the proposed packing strategy; $\mathcal{T}$ should be $O((n \log n)/g)$ to achieve a speedup of g with g machines.

A primitive operation we need is sorting. In the MPC model, sorting n elements (initially evenly distributed on the g machines) can be done in $O(\log_s n)$ rounds, $O((n \log n)/g)$ running time, $O(n \log n)$ total amount of computation, and $O((n \log_s n)/g)$ load (communication I/Os) [24] (see Tao et al. [57] for a simple algorithm when $s \geq g \ln(g \cdot n)$ holds).

Mapping n data points to the rank space and sorting them by their Z-order values thus can be done in $O(\log_s n)$ rounds. This process takes $O((n \log n)/g)$ running time and $O(n \log n)$ total amount of computation. We focus on packing the sorted data points to form an R-tree next.

4.2 Distributed Packing

Every round bulk-loads $\Theta(\log_B s)$ levels of the target R-tree. In the first round, $O(s)$ consecutive data points are assigned to a machine by the ascending order of their Z-order values, where an R-tree of $\Theta(\log_B s)$ levels is bulk-loaded locally. This creates $O(n/s)$ R-trees. A second round bulk-loads the next $\Theta(\log_B s)$ levels of the target R-tree over the root MBRs of those $O(n/s)$ R-trees. For this purpose, $O(1 + g/s)$ machines are used, each assigned $O(s)$ root MBRs; this results in $O(n/s^2)$ tree roots. The above process repeats until the MBRs can all be bulk-loaded in a single machine (the number of participating machines decreases by a factor of $\Theta(s)$ each time, while each such machine is always assigned $O(s)$ MBRs). A total of $O(\log_B n/\log_B s) = O(\log_s n)$ rounds are incurred, where $O(s \log_s n) = O((n \log_s n)/g)$ running time and $O(n)$ total amount of computation are taken to compute the MBRs. Over the $O(\log_s n)$ rounds, the maximum load of any participating machine, i.e., the load (communication I/O cost) of our packing algorithm, is also $O(s \log_s n) = O((n \log_s n)/g)$.

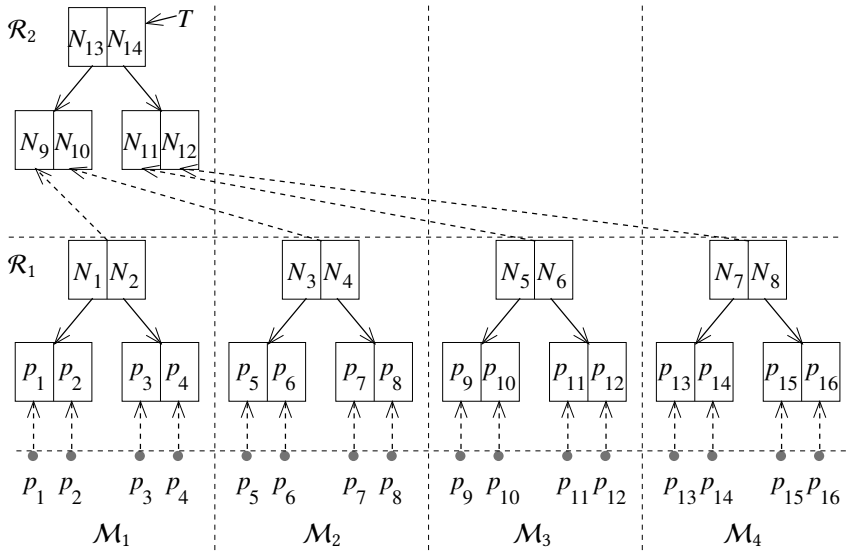


Fig. 6. Parallel R-tree bulk-loading

The rounds are illustrated in Fig. 6, where $n = 16$, $B = 2$, $g = 4$, and $s = 4$. A total of $\log_s n = 2$ rounds are needed. Each round bulk-loads $\log_B s = 2$ levels. In the first round $\mathcal{R}_1$, every machine is assigned $s = 4$ data points. The 4 machines bulk-load 4 R-trees of 2 levels locally. The 4 MBRs of the roots of these local R-trees are bulk-loaded by a single machine $\mathcal{M}_1$ in the second round $\mathcal{R}_2$.

We omit the pseudo-code of the parallel bulk-loading algorithm as it is similar to Algorithm 1, except that now a machine handles $O(s)$ data points instead of n, and the loop to bulk-load an R-tree (Lines 9 to 12) is broken into rounds.

5 UPDATE HANDLING

In previous sections, we focused on bulk-loading a static R-tree structure to guarantee a worst-case optimal window query performance. In this section, we discuss how to handle data updates for the bulk-loaded tree without impacting the worst-case optimal query performance. We will first convert the static R-tree structure into a *deletion-only* R-tree structure in Section 5.1. This structure retains the $O(n/B)$ space cost and the $O((n/B)^{1-1/d} + k/B)$ window query I/O cost, while it can also handle a deletion in $O(\log_B n)$ amortized I/Os. The structure does not support insertions. We then extend this structure to support insertions in Section 5.2, which leads to a *fully dynamic* structure

named the *LogR-tree* that still answers a window query in $O((n/B)^{1-1/d} + k/B)$ I/Os. We study how to further reduce the insertion cost, resulting in an insertion improved structure named the *LogR*-tree* in Section 5.3.

5.1 Deletion

Recall that our static R-tree structure stores all the points of P sorted by their Z-order values. In this tree, an inner node entry stores a pointer to a child node and its MBR. We modify this tree slightly such that an inner node entry also stores the minimum Z-order value of the corresponding child node. This increases the size of an inner node entry and reduces the node capacity B by a small constant factor, but it does not impact either the space cost or the window query cost in big-O notation. The modified R-tree structure can be seen as a B-tree Γ over P with the Z-order values as the key values. This structure is illustrated in Fig. 7a, where the numbers in parenthesis represent Z-order values, e.g., p_2 has a Z-order value of 1, and N_1 has a minimum Z-order value of 1.

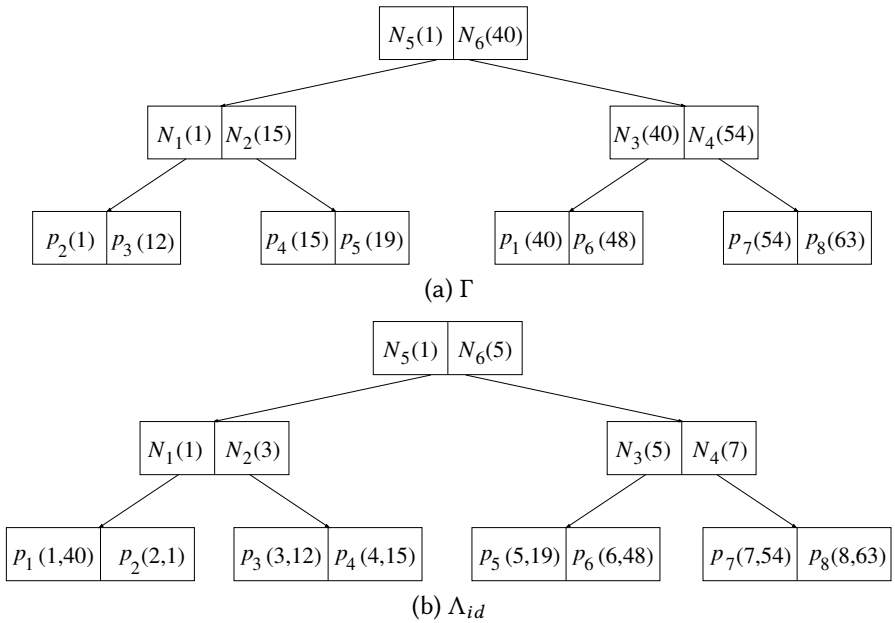


Fig. 7. Deletion-only structure

We support deletion by object identifiers. Every point $p \in P$ is associated with a *unique integer identifier* denoted by id, and a user supplies the id of p to trigger the deletion of p. Since the B-tree Γ is constructed using Z-order values as the keys, we need to further construct a structure to map id's to Z-order values and enable deletion by id. This is done by an additional B-tree over the points in P, where the id's are the keys, and the points in P are stored in the leaf nodes with their Z-order values. We call this additional B-tree the *ID B-tree* and denote it by Λ_{id}. As Fig. 7b shows, the ID B-tree stores the data points in its leaf nodes. Each point is associated with a pair (id, z_value) representing the id and the Z-order value of the point, e.g., p_2 has an id of 2 and a Z-order value of 1. The minimum data point id in a leaf node is stored in its corresponding entry in the parent node, e.g., the minimum data point id of the leftmost leaf node, node N_1, is 1, which is stored in the parent node as $N_1(1)$.

Set $n_{last} = n$ right after constructing the B-trees Γ and Λ_{id}. To delete a point p, we compute its Z-order value by a point search over Λ_{id} and delete its entry from both Γ and Λ_{id} in $O(\log_B n)$

I/Os. This may trigger an underflow in the node containing p, which is handled in the same way as a normal B-tree. We decrease n by 1. If n has dropped to $n_{last}/2$, we perform an overhaul, which destroys the B-trees Γ and Λ_{id}, reconstructs two new ones from (the remaining points in) P, and resets $n_{last} = |P|$.

Cost analysis. It is clear that the space occupied by the deletion-only structure is $O(n/B)$ at all times (n equals to the size of the current P).

Regarding the deletion cost, first note the trivial fact that the cost is $O(\log_B n)$ if an overhaul does not take place. If an overhaul happens, it takes $O(sort(n)) = O((n/B)\log_{M/B}(n/B))$ I/Os where $M \geq 2B$ is the memory size (in number of words). We charge the cost over the $n_{last}/2 = n$ deletions since the last overhaul. Therefore, each deletion bears only $O((1/B)\log_{M/B}(n/B)) = o(\log_B n)$ I/Os. Note the little-o notation here. Thus, the amortized deletion cost is bounded by $O(\log_B n)$.

It remains to prove that the query cost is $O((n/B)^{1-1/d} + k/B)$. Using precisely the same argument as in the proof of Lemma 3.1 (and the discussion on the case where n is not a power of 2) in Section 3.3.1, we know that the query cost is bounded by $O((n_{last}/B)^{1-1/d} + k/B)$. This is $O((n/B)^{1-1/d} + k/B)$ because $n \geq n_{last}/2$.

5.2 Insertion

We now combine the deletion-only structure with the *logarithmic method* [13, 45] to obtain a fully dynamic structure. The key idea of the logarithmic method is to replace insertions by constructing a series of new deletion-only structures to hold the points to be inserted. There is no real insertion occurring on any of the deletion-only structures constructed (except for insertions on the Λ_{id} tree). Thus, the deletion and query cost bounds of the deletion-only structures are preserved.

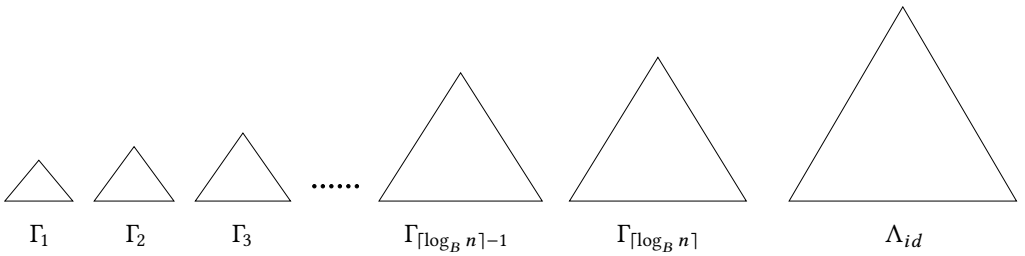


Fig. 8. LogR-tree

Specifically, as shown in Fig. 8, we create a series of $\lceil \log_B n \rceil$ deletion-only structures denoted by $\Gamma_1, \Gamma_2, \ldots, \Gamma_{\lceil \log_B n \rceil}$, where Γ_i can index up to B^i data points. The initial data set P is indexed in $\Gamma_{\lceil \log_B n \rceil}$, while the rest of the deletion-only structures are empty.

To insert a new point p_{new}, we find the smallest j such that $1 + \sum_{i=1}^{j} |\Gamma_i| \leq B^j$, where $|\Gamma_i|$ represents the number of points indexed in Γ_i. We use p_{new} and the points in $\Gamma_1, \Gamma_2, \ldots, \Gamma_j$ to bulk-load a new Γ_j, and empty $\Gamma_1, \Gamma_2, \ldots, \Gamma_{j-1}$. To delete a point p_{del}, we locate the deletion-only structure that indexes p_{del} and delete p_{del} following the procedure described in Section 5.1. To help locate p_{del}, we modify the ID B-tree Λ_{id} such that it also stores the id of the deletion-only structure in which a point is indexed, denoted by $tree\_id$. Thus, in Λ_{id}, a leaf node entry now stores a triple $(id, tree\_id, z\_value)$ instead of (id, z_value). A window query is processed over each of $\Gamma_1, \Gamma_2, \ldots, \Gamma_{\lceil \log_B n \rceil}$, and the results are combined together as the final query answer.

We use *LogR-tree* to denote the above structure resulted from applying the logarithmic method over our deletion-only structure. The following result holds for the LogR-tree, which is due to Arge and Vahrenhold [8].

Lemma 5.1. *Suppose that we have an $O(n/B)$-space structure that supports a deletion in $O(\log_B n)$ amortized I/Os, answers a reporting query in $O(Q(n) + k/B))$ I/Os (k is the number of points reported),*

and can be constructed in $O(sort(n))$ I/Os. Then, we can apply the logarithmic method to obtain a structure with the following guarantees:

- *Space cost:* $O(n/B)$*;*
- *Query cost:* $O\Big(\sum_{i=1}^{\lceil \log_B n \rceil} Q(\min\{B^i, n\})\Big) + O(k/B)$*;*
- *Deletion cost:* $O(\log_B n)$ *amortized;*
- *Insertion cost:* $O(\log_B^2 n + \log_B n \cdot \log_{M/B}(n/B))$ *amortized.*

Applying the lemma to our deletion-only structure described in Section 5.1, we have $Q(n) = (n/B)^{1-1/d}$. The lemma yields a fully-dynamic structure that consumes $O(n/B)$ space, and supports a deletion in $O(\log_B n)$ amortized I/Os and an insertion in $O(\log_B^2 n + \log_B n \cdot \log_{M/B}(n/B))$ amortized I/Os. The query cost is bounded by:

$$\begin{aligned} & O\Big(\sum_{i=1}^{\lceil \log_B n \rceil} Q(\min\{B^i, n\})\Big) + O(k/B) \\ &= O\Big(\sum_{i=1}^{\lceil \log_B n \rceil} \min\{(B^i/B)^{1-1/d}, (n/B)^{1-1/d}\}\Big) + O(k/B) \\ &= O((n/B)^{1-1/d} + k/B) \end{aligned} \tag{4}$$

Note that the summation term in the equation above is asymptotically dominated by the last term (i.e., $i = \lceil \log_B n \rceil$).

Nowadays, the memory size M typically satisfies $\log_{M/B}(n/B) = O(1)$ – recall that $O(\log_{M/B}(n/B))$ is the number of passes performed by an external sort on n points. The insertion cost is then $O(\log_B^2 n)$ amortized.

5.3 Improving the Insertion Cost

Next, we will hack into the logarithmic method and present an improved version of Lemma 5.1 specific to our structures. The improvement lowers the amortized insertion cost from $O(\log_B^2 n)$ to $O(\log_B n)$ when $\log_{M/B}(n/B) = O(1)$.

The description below essentially follows the ideas of Arge and Vahrenhold [8], but introduces new pointers to lower the insertion cost. The focus will be placed on explaining the algorithm steps involving these pointers and the corresponding analysis.

B-tree. Regarding the B-trees used in our structure, we require that:

- All the data points are at the leaf level.
- If a leaf node overflows, $\Omega(B)$ points must have been inserted into the node since the node was created.
- If a leaf node underflows, $\Omega(B)$ points must have been deleted from the node since the node was created.

These requirements can be easily fulfilled by slightly modifying the standard B-tree algorithms (see, e.g., Arge and Vitter [9] and Huddleston and Mehlhorn [29]).

The LogR*-tree structure. We use *LogR*-tree* to denote the proposed structure with improved insertion costs. The LogR*-tree resembles the LogR-tree as shown in Fig. 8, but with additional pointers in the leaf nodes of the deletion-only structures Γ_i. For conciseness, we do not draw another figure to illustrate the LogR*-tree.

In a LogR*-tree, at all times, the input set P is stored in a sequence of $H \leq 1 + \lceil \log_B n \rceil$ deletion-only structures $\Gamma_1, \Gamma_2, \ldots, \Gamma_H$ that satisfy the following conditions.

- Each point in P is in one and only one deletion-only structure;
- The number of points in Γ_i, denoted by $|\Gamma_i|$, can be anywhere from 0 to B^i.

Each point $p \in P$ is said to have the structure index i if p is stored in Γ_i.

Same as that in the LogR-tree, the ID B-tree Λ_{id} in LogR*-tree also stores the structure index of the points. For each point $p \in P$, its entry in Λ_{id} stores its *id*, structure index $tree\_id$, and Z-order value in the structure z_value. Now Λ_{id} serves as a "dictionary" that maps the *id* of a point to its structure index in $O(\log_B n)$ I/Os.

Based on the structural design above, each point $p \in P$ with structure index i is stored: (i) *at a leaf node u of Γ_i* and (ii) *at a leaf node v of Λ_{id}*.

We refer to the address of v as the *dictionary address* of p. Along with the entry of p in u, we store a pointer to v, which we call the *dictionary pointer* of p. As an example, consider the structure Γ and its corresponding ID B-tree Λ_{id} in Fig. 7. The dictionary pointer of p_2 should point to N_1 of Λ_{id}, since p_2 is in N_1 of Λ_{id}. This pointer allows us to fetch v in a single I/O once u has been found, which is essential for reducing the insertion cost from $O(\log_B^2 n)$ to $O(\log_B n)$.

Finally, we also store an integer n_{global} to be defined in the global rebuilding operation below.

Global rebuilding. This operation constructs a "clean" structure from the current P with n points. It takes the following steps:

- Destroy all structures.
- Build (i.e., bulk-load) a deletion-only structure $\Gamma_{\lceil \log_B n \rceil}$ on all the points in P.
- Build Λ_{id} and update the dictionary pointers in $\Gamma_{\lceil \log_B n \rceil}$.
- Set n_{global} to n.

It is rudimentary to implement the above operation in $O(sort(n)) = O((n/B)\log_{M/B}(n/B))$ I/Os. We use the above operation to build the first structure from the initial P (before all updates). In general, after a global rebuild, we perform the next global rebuild after $\lceil n_{global}/2 \rceil$ updates. The global rebuild takes $O(sort(n_{global})) = O((n_{global}/B)\log_{M/B}(n_{global}/B))$ I/Os. Therefore, each of those updates is amortized only $o(\log_B n)$ I/Os.

Query. To answer a query, we simply search all the H deletion-only structures in the LogR*-tree. The query cost is:

$$O\Big(\sum_{i=1}^{1+\lceil \log_B n \rceil} Q(\min\{B^i, n\})\Big) + O(k/B) = O((n/B)^{1-1/d} + k/B) \tag{5}$$

Algorithm 2: LogR*-tree-Deletion

Input: $\langle \Gamma_1, \Gamma_2, \ldots, \Gamma_H; \Lambda_{id} \rangle$: a LogR*-tree; p_{del}: a point to be deleted.
Output: $\langle \Gamma_1, \Gamma_2, \ldots, \Gamma_H; \Lambda_{id} \rangle$: the updated LogR*-tree.

1 $i, z \leftarrow$ Point query on Λ_{id} to find p_{del}, return tree id i and Z-order value z of p_{del};
2 Delete p_{del} from Γ_i using standard B-tree deletion procedures by key value z;
3 Delete p_{del} from Λ_{id} using standard B-tree deletion procedures;
4 **if** *underflow and node merging occur in* Λ_{id} **then**
5 **for** *each* $p \in \Lambda_{id}$ *that has been moved to a new node* v **do**
6 $i', z' \leftarrow$ tree id i' and Z-order value z' of p;
7 Point query on $\Gamma_{i'}$ by key value z' to find p;
8 Dictionary address of $p \leftarrow v$;
9 **return** $\langle \Gamma_1, \Gamma_2, \ldots, \Gamma_H; \Lambda_{id} \rangle$;

Deletion. We delete a point p_{del} in two steps as summarized in Algorithm 2:

(1) Find the structure index i of p_{del} using Λ_{id} and perform the deletion in Γ_i (Lines 1 and 2). This takes $O(\log_B n)$ I/Os.

(2) Delete p_{del} from Λ_{id} (Line 3). If this triggers an underflow, treating the underflow may change the dictionary addresses of $O(B)$ points in the deletion-only structures. For every such point p, we perform a point query on its corresponding deletion-only structure to locate it and update its dictionary pointer (Lines 4 to 8), which takes $O(\log_B n)$ I/Os. A total of $O(B \log_B n)$ I/Os are incurred by these dictionary pointer updates. However, an underflow can happen only after $\Omega(B)$ points have been deleted from the node. We can charge the $O(B \log_B n)$ cost over those deletions, each of which is amortized only $O(\log_B n)$ I/Os.

Insertion. We insert a point p_{new} as follows, which is summarized in Algorithm 3.

(1) Find the smallest j satisfying $1 + \sum_{i=1}^{j} |\Gamma_i| \leq B^j$ (recall that $|\Gamma_i|$ is the number of points stored in Γ_i, Lines 1 to 5). Denote by S_1 the set of points stored in $\Gamma_1, \Gamma_2, \ldots, \Gamma_{j-1}$ and denote by S_2 the set of points in Γ_j. Destroy $\Gamma_1, \Gamma_2, \ldots, \Gamma_j$, and construct a new Γ_j on $S_1 \cup S_2 \cup \{p_{new}\}$ (Lines 6 to 11). This process takes $O(sort(B^j)) = O(B^{j-1} \log_{M/B} B^{j-1}) = O(B^{j-1} \log_{M/B}(n/B))$ I/Os. For every point $p \in S_1$, update its structure index in Λ_{id} (Lines 7 and 8). This takes only $O(1)$ I/Os using the dictionary pointer of p, which results in $O(|S_1|)$ I/Os in total. Every such p has moved up to a deletion-only structure with a higher index – we say that p has been *promoted*. By definition of j, we have $|S_1| \geq B^{j-1}$. Hence, the total cost of this step is $O(|S_1| \log_{M/B}(n/B))$ I/Os. We charge this cost over the points in S_1 such that every promoted point is amortized $O(\log_{M/B}(n/B))$ I/Os.

(2) Insert p_{new} into Λ_{id} (Line 12). If this triggers an overflow, treating the overflow may change the dictionary addresses of $O(B)$ points in the deletion-only structures. For every such point, updating its dictionary pointer takes $O(\log_B n)$ I/Os (Lines 13 to 17). A total of $O(B \log_B n)$ I/Os are incurred by these dictionary pointer updates. However, an overflow can happen only after $\Omega(B)$ points have been inserted into the node. We can charge the $O(B \log_B n)$ cost over those insertions, each of which is amortized only $O(\log_B n)$ I/Os.

Algorithm 3: LogR*-tree-Insertion

Input: $\langle \Gamma_1, \Gamma_2, \ldots, \Gamma_H; \Lambda_{id} \rangle$: a LogR*-tree; p_{new}: a point to be inserted.
Output: $\langle \Gamma_1, \Gamma_2, \ldots, \Gamma_H; \Lambda_{id} \rangle$: the updated LogR*-tree.

```
1  i ← 1, sum ← |Γ_i|;
2  while 1 + sum > B^i do
3      i ← i + 1;
4      sum ← sum + |Γ_i|;
5  j ← i;
6  S_1 ← the set of points in Γ_1, Γ_2, ..., Γ_{j-1};
7  for each p ∈ S_1 do
8      Find p in Λ_id via its dictionary pointer and update its tree id to j;
9  S_2 ← the set of points in Γ_j;
10 Destroy Γ_1, Γ_2, ..., Γ_j;
11 Bulk-load Γ_j with S_1 ∪ S_2 ∪ {p_new};
12 Insert p_new into Λ_id using standard B-tree insertion procedures;
13 if overflow and node split occur in Λ_id then
14     for each p ∈ Λ_id that has been moved to a new node v do
15         i', z' ← tree id i' and Z-order value z' of p;
16         Point query on Γ_{i'} by key value z' to find p;
17         Dictionary address of p ← v;
18 return ⟨Γ_1, Γ_2, ..., Γ_H; Λ_id⟩;
```

Finishing the update cost analysis. Suppose that we perform μ updates in total. Let n_i be the value of n before the i-th ($1 \leq i \leq \mu$) update. We prove that our algorithms handle these updates in $O(\sum_{i=1}^{\mu}(1 + \log_B n_i \cdot \log_{M/B}(n_i/B)))$ I/Os. This proves that our amortized cost is $O(\log_B n \cdot \log_{M/B}(n/B))$ per insertion and deletion.

We will focus on bounding the cost that arises at Step (1) of the insertion procedure. By the earlier discussion, it is clear that the other steps in the insertion and deletion procedures perform $O(\log_B n)$ amortized I/Os per insertion and deletion.

The cost of Step (1) of insertion can be computed via the number of point promotions, since every point promotion bears an amortized I/O cost of $O(\log_{M/B}(n/B))$. The number of point promotions is in turn determined by the number of points promoted and the number of times that these points are promoted. We analyze these two factors by separating the μ updates into epochs.

Suppose that there were η global rebuilds in total. They divide the time line into η epochs, where the j-th epoch starts from the moment when the j-th global rebuild happened and ends right before the next global rebuild (the last epoch is "open" by this definition). Define $\widetilde{n}_j$ as the value of n at the j-th global rebuild ($\widetilde{n}_1$ is the size of the initial P before all updates).

Since there are at most $\lceil \widetilde{n}_j/2 \rceil$ updates in the j-th epoch, the number of points that were promoted at least once in this epoch is obviously $O(\widetilde{n}_j)$. A point can only be promoted to a deletion-only structure with a higher index. Thus, the number of deletion-only structures H in the LogR*-tree in the j-th epoch bounds the number of times that a point can be promoted. The value of H is bounded by the following lemma.

Lemma 5.2. *The number of deletion-only structures in the LogR*-tree (i.e., the value of H) remains between $\lceil \log_B \widetilde{n}_j \rceil$ and $1 + \lceil \log_B \widetilde{n}_j \rceil$ throughout the j-th epoch.*

Proof. The value of H equals $\lceil \log_B \widetilde{n}_j \rceil$ right after the j-th global rebuild and never decreases (recall that $\Gamma_{\lceil \log_B \widetilde{n}_j \rceil}$ has $\widetilde{n}_j$ points at the beginning of the j-th epoch, while the epoch has at most $\widetilde{n}_j/2$ deletions). Meanwhile, at any moment, it must hold that $B^H \leq n \leq 3\widetilde{n}_j/2$ since we perform global rebuilding after $\lceil \widetilde{n}_j/2 \rceil$ updates. Hence, $H \leq \log_B(3\widetilde{n}_j/2) \leq 1 + \lceil \log_B \widetilde{n}_j \rceil$ because $B \geq 2$. □

Based on Lemma 5.2, in the j-th epoch, a point can be promoted $O(\log_B \widetilde{n}_j)$ times. As discussed above, there are $O(\widetilde{n}_j)$ points promoted where each promotion bears an amortized I/O cost of $O(\log_{M/B}(n/B))$. Thus, Step (1) of insertion incurs in total $O(\widetilde{n}_j \cdot \log_B \widetilde{n}_j \cdot \log_{M/B}(n/B))$ I/Os. This means $O(\log_B \widetilde{n}_j \cdot \log_{M/B}(n/B))$ I/Os per update.

Therefore, we obtain a fully dynamic structure that has $O(n/B)$ space cost, $O((n/B)^{1-1/d} + k/B)$ query I/O cost, $O(\log_B n)$ deletion I/O cost, and $O(\log_B n \cdot \log_{M/B}(n/B))$ insertion I/O cost. As mentioned earlier, the insertion I/O cost becomes $O(\log_B n)$ when $\log_{M/B}(n/B) = O(1)$.

5.4 Practical Considerations

In this section, we constructed a dynamic structure named LogR*-tree that retains the worst-case optimal query performance while also having attractive update performance. Due to the complex design of this structure, there are factors to be considered when applying this structure.

In terms of the application scenarios, as mentioned in Section 1, we target applications such as digital mapping where queries are much more frequent than updates over the data, e.g., there may be millions of users querying Google Maps while the map data may not require constant updates. Our index design is thus prioritized for the query performance. Our static structure offers an empirically efficient and worst-case optimal query performance. It may be used for applications such as data warehousing that allow periodic rebuilds (e.g., overnight). Our dynamic structure (i.e., the LogR*-tree) further allows online data updates for applications such as digital mapping without impacting the worst-case optimal query performance, which is the core contribution of

Section 5. However, we do acknowledge that our LogR*-tree is not designed for applications with a high data update frequency, e.g., moving object databases. How to retain the worst-case optimal query performance in such applications is challenging and an interesting future study. Also, we focus on relatively low dimensional space, and mention that there are studies for high-dimensional window queries (e.g., [64]), but those are not our target applications.

In terms of the implementation complexity, while our LogR*-tree may look complex at a first glance, it only consists of slightly adapted versions of B-trees and can be implemented based on the B-tree. Index management over our LogR*-tree such as concurrency control and caching can be done using standard B-tree concurrency control and caching algorithms. Our use of the ID B-tree to keep track of the tree IDs of the data points does bring extra update workloads, and it breaks the nice property of being cache-oblivious for the trees constructed by the logarithmic method. These may impact the update efficiency and the throughput of the database system. As our experiments in Section 6.2.3 show, our LogR*-tree does have a higher update cost than a baseline tree structure that uses the logarithmic method without the ID B-tree. This would limit the applicability of our index structure to applications with highly frequent updates as discussed above.

In terms of the update costs, they are achieved based on an amortized analysis. A global rebuild is needed after every $n/2$ updates, which can bring I/O peaks. To avoid impacting query users' experience, an update server (or a cluster) may run in parallel with the query server to perform global rebuilds. During a rebuild, the query server can query the "old" index structure and scan the data points updated after the rebuild is triggered (which should not be many in our target applications) to provide query answers. There are also techniques (e.g., [46, 47, 53]) to de-amortize the costs for the logarithmic method. Their basic idea is to distribute the workload of a global rebuild across the updates in a rebuild cycle, such that no updates have a significantly higher workload than the others. However, these techniques are mainly of theoretical interest. Adapting them for our dynamic structures to achieve an empirically efficient de-amortization while preserving the theoretical optimality would require significant research efforts. We leave this task for future study.

6 EXPERIMENTS

We study the empirical performance of the proposed algorithms in this section.

6.1 Experimental Setup

Algorithms. As summarized in Table 2, we test the following algorithms. For the bulk-loading and window query processing performance, we compare our bulk-loading algorithm with the STR-tree [36], Hilbert R-tree [28], H-GO R-tree [1], TGS R-tree [23], and PR-tree [7], which have been described in Section 2. Note that the H-GO R-tree assumes known query width and height in its original proposal [1]. We adapt it by ignoring the query width and height (i.e., letting them be 0) to keep consistency with the rest of the algorithms. We denote the baseline algorithms by "**STR**", "**HR**", "**HGO**", "**TGS**", and "**PR**", respectively. We denote the proposed algorithm by "**ZR**" for that it builds an R-tree based on Z-order values.

As discussed in Section 3.4, the proposed packing strategy is also applicable to other space-filling curves such as the Hilbert curve. To demonstrate this applicability, we further implement an R-tree based on the proposed packing strategy where the data points are sorted and packed by their Hilbert-order values in the rank space. We denote this R-tree by "**HRR**". This R-tree shares a similar structure with the Hilbert R-tree, except that the data points are mapped to the rank space before they are packed. Note that the query cost bounds derived in Section 3.3 hold for this tree.

For the update handling performance, we compare our LogR-tree and LogR*-tree with the LR-tree [16] that applies the logarithmic method over the Hilbert R-tree as discussed in Section 2. We denote this baseline algorithm by "**LR-tree**". We implement the LogR-tree and the LogR*-tree

Table 2. Algorithms Evaluated

Experiment	Group	Algorithm	Description
Bulk-loading and query processing	Baseline	HGO	H-GO [1]
		HR	Hilbert R-tree [28]
		PR	PR-tree [7]
		STR	STR-tree [36]
		TGS	TGS R-tree [23]
	Proposed	HRR	Rank space technique (using Hilbert curve)
		ZR	Rank space technique (using Z-curve)
Parallel bulk-loading	Baseline	L-C	Level-by-level technique (communication time)
		L-M	Level-by-level technique (running time $\mathcal{T}$)
		L-R	Level-by-level technique (response time)
	Proposed	ZR-C	Multi-level technique (communication time)
		ZR-M	Multi-level technique (running time $\mathcal{T}$)
		ZR-R	Multi-level technique (response time)
Update handling	Baseline	LR-tree	LR-tree [16]
	Proposed	LogR-H	LogR-tree (using Hilbert curve)
		LogR-Z	LogR-tree (using Z-curve)
		LogR*-H	LogR*-tree (using Hilbert curve)
		LogR*-Z	LogR*-tree (using Z-curve)

over both Z-curves and Hilbert-curves. We denote the resultant trees by "**LogR-Z**", "**LogR*-Z**", "**LogR-H**", and "**LogR*-H**", where the suffixes "-Z" and "-H" denote Z-curve and Hilbert-curve based implementations, respectively. We do not compare with the other baseline bulk-loading algorithms (HGO, PR, STR, and TGS) because their bulk-loaded R-trees are uncompetitive in query processing, as shown in the experimental results in Section 6.2.1. Combining such R-trees with the logarithmic method to handle updates will not be competitive either. We also note that the PR-tree has an update algorithm [7] based on the logarithmic method. However, this algorithm is more of theoretical interest. No implementation or empirical result has been presented for it.

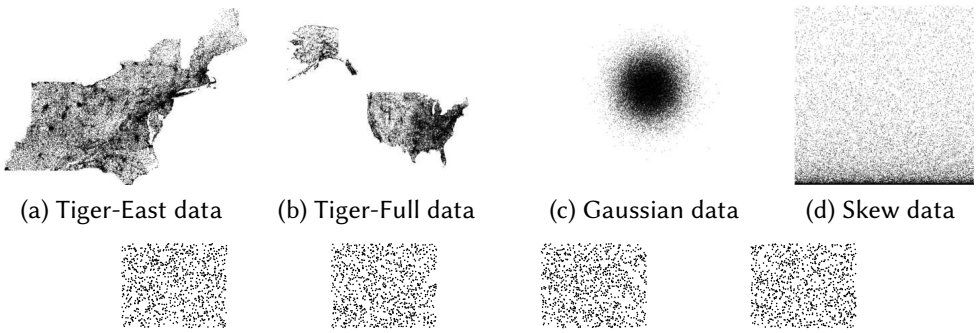
Following previous studies [7, 23, 28, 36], we focus on the I/O cost of the algorithms above.

For the parallel bulk-loading performance, we compare our proposed multi-level algorithm with a level-by-level parallel bulk-loading algorithm that also uses the proposed packing strategy. We implement these algorithms over both Z-curves and Hilbert curves. We observe that the SFC used has very small impact on the parallel bulk-loading performance, since it only affects the curve values of the data points. To keep the figures concise, we only report the results over Z-curves and denote the two corresponding algorithms by "**ZR**" and "**L**", respectively. For these parallel algorithms, we measure (i) the response time (denoted by "**ZR-R**" and "**L-R**", respectively), which is the duration for which an algorithm runs, (ii) the running time $\mathcal{T}$ (denoted by "**ZR-M**" and "**L-M**", respectively), which is the sum of the maximum single machine response time over all MapReduce rounds of an algorithm, and (iii) the communication time (denoted by "**ZR-C**" and "**L-C**", respectively), which is the part of the response time spent on communication. We do not measure the I/O cost of the parallel algorithms because they are based on Spark which has a different I/O mechanism from those of the standalone algorithms based on the TPIE library [37].

System environment. The window query and index update experiments are run on a 64-bit machine running Ubuntu 14.04 with a 2.60 GHz Intel i5 CPU, 4 GB memory, a 1 TB TOSHIBA MQ01ABD075 (5400 RPM) hard disk drive, and a 240 GB SanDisk SSD Plus solid-state drive. We use

Ke Yi's single-machine implementation[1] of the Hilbert R-tree, TGS R-tree, and PR-tree, which uses the TPIE library [37] – a C++ library that provides APIs for implementation of external memory algorithms and data structures. For ease of comparison, we also implement a single-machine version of the H-GO R-tree (based on Ke Yi's Hilbert R-tree implementation), the STR-tree, the LR-tree, and the proposed HRR, ZR, LogR-tree and LogR*-tree using TPIE. In all the R-tree structures except for those used in the LogR-tree and the LogR*-tree, we use 40 bytes for each entry in a node. For an inner node entry, these 40 bytes include 32 bytes for the 4 coordinates (8 bytes each) of an MBR and 8 bytes for a pointer pointing to the disk block storing the corresponding child node. For a leaf node entry, these 40 bytes include 8 bytes for an id of a data point and 32 bytes for the coordinates also in the form of an MBR for ease of implementation. For the LogR-tree, each entry in an R-tree node has 48 bytes instead of 40, where the 8 extra bytes store a space-filling curve value. For the LogR*-tree, a leaf node entry needs to additionally store a space-filling curve value (8 bytes) and a dictionary pointer (8 bytes). We thus use 56 bytes for each entry. We use a block size of 4 KB. This means that the maximum fanout of an R-tree node (i.e., B) is 102 (85 for the LogR-tree and 73 for the LogR*-tree). For the H-GO R-tree, following its original proposal [1], we use $B/3$ to bound the minimum number of entries in an R-tree node.

The bulk-loading experiments are run on the single machine described above and on a cluster. The parallel bulk-loading algorithms are implemented with Scala and run on Apache Spark 1.6.0-SNAPSHOT, which also supports the MapReduce model but is more efficient than Hadoop MapReduce. We use a cluster with 16 virtual nodes from an academic computing cloud [40] running on OpenStack. Each virtual node has 12 GB memory and 4 cores running at 2.6 GHz. One of the virtual nodes acts as the master and the other 15 virtual nodes act as slaves. Each core simulates a worker machine, and hence there are 60 worker machines in total, i.e., $g = 60$. The network bandwidth is up to 200 Mbps. We use Apache Hadoop 2.6.0 with Yarn as the resource manager.



(a) Tiger-East data (b) Tiger-Full data (c) Gaussian data (d) Skew data

(e) Cluster data

Fig. 9. Experimental data

Data sets. We use both real and synthetic data sets. The real data set contains 17,468,292 rectangles (666 MB in size) representing geographical features in 18 eastern states of the USA extracted from a subset of the TIGER/Line 2006SE data [59]. We use the center of the rectangles as our data points. We denote this data set by "**Tiger-East**" and plot it in Fig. 9a. For the parallel bulk-loading experiments, we further construct a data set that contains the rectangle centers from the full TIGER/Line 2006SE data (62,174,885 rectangles, 2.32 GB in size). We denote it by "**Tiger-Full**" and plot it in Fig. 9b.

Synthetic data sets are generated with a space domain of 1×1 where the data set cardinality ranges from 0.5 to 20 million (and up to 100 million for parallel bulk-loading experiments). We generate

[1]https://www.cse.ust.hk/~yike/prtree/

Table 3. Parameters and Their Settings

Parameter	Setting
Data sets	Tiger-East, Tiger-Full, Uniform, Gaussian, Skew, Cluster
d	**2**, 3, 4, 5
n (million)	0.5, 1, 5, **10**, 20, 40, 60, 80, 100
Cache size (blocks)	**0**, 1, 4, 16, 64, 256
Percentage of data updates (%)	20, 40, 60, **80**, 100, 120
Query window area (%)	0.0001, 0.001, **0.01**, 0.1, 1, 2
Storage hardware	**hard disk drive**, solid-state drive

four groups of synthetic data sets, denoted by "**Uniform**", "**Gaussian**", "**Skew**", and "**Cluster**", respectively. The Uniform and Gaussian data sets follow uniform and Gaussian distributions ($\mu = 0.5$ and $\sigma = 1$), respectively. We plot a sample Gaussian data set in Fig. 9c. The Skew and Cluster data sets are generated following the PR-tree paper [7]. A Skew data set is generated from a Uniform data set by raising the y-coordinates to their powers, i.e., the coordinates of a randomly generated data point are converted from (x, y) to (x, y^α), $\alpha = 9$. We plot a sample Skew data set in Fig. 9d. The Cluster data set is designed to test the worst-case window query performance of the R-trees bulk-loaded using an SFC. It contains 10,000 clusters with centers evenly distributed on a horizontal line. Each cluster contains a subset of points following a uniform distribution in a 0.00001×0.00001 square around the center. We plot a sample Cluster data set with four clusters in Fig. 9e.

We vary the query window size, the data set size, the data dimensionality, the cache size, the hardware for index storage, and the percentage of data updates. The experimental parameters are summarized in Table 3, where default values are in bold.

6.2 Results

We present results on window query processing, bulk-loading, and update handling, respectively.

6.2.1 Window Query Processing. We start with the window query performance of the bulk-loaded R-trees (without data updates). We generate 100 square-shaped queries at locations following the data distribution in each experiment except for the experiments on the Cluster data set. The Cluster data set is designed to test the worst-case performance of the R-trees. Following the PR-tree paper [7], we generate long and thin window queries to query this data set. The bottom-left (bottom-right) corner of each query is randomly placed to the left (right) of the leftmost (rightmost) cluster, such that the query spans all 10,000 clusters. The height of the query is generated as the intended query window size divided by the query width.

For ease of comparison, we follow previous studies [7, 23, 28, 36] and report the average I/O cost per query relative to the output size. Let the number of blocks read for a query be I and the output size be k/B. We report $I/(k/B)$. Note that $I/(k/B) \geq 1$, i.e., we need to at least read all the blocks containing the data points in the query answer. A smaller value of $I/(k/B)$ is more preferable.

Varying the query window size. We first vary the area of the query window from 0.0001% to 2% of the data space. We show the query I/O cost relative to the output size k/B over 10 million data points (17 million for Tiger-East) in Fig. 10. A general observation is that the relative query costs of the R-trees decrease as the query window area increases. This is because a larger query window overlapping a tree node is more likely to overlap the data points in this node, i.e., there are lower percentages of extra query I/Os that do not contribute to the output.

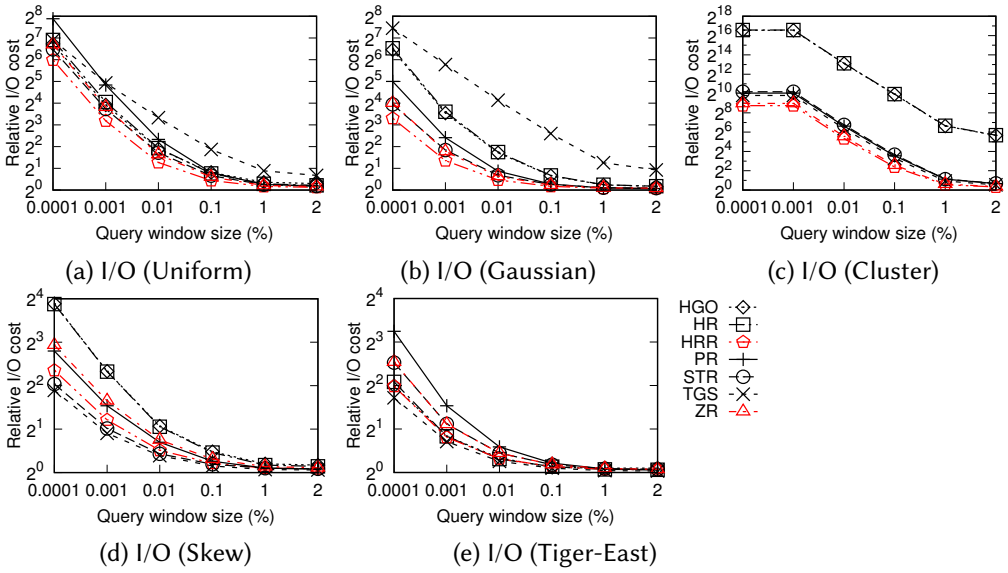


Fig. 10. Query costs – varying the query window size

The R-trees HRR and ZR created by the proposed packing strategy have the smallest query I/O costs on Uniform, Gaussian, and Cluster data (Figs. 10a, 10b, and 10c). On Skew and Tiger-East data, the query I/O costs of HRR and ZR are close to those of TGS which are the smallest (Figs. 10d and 10e). This demonstrates that HRR and ZR not only have an asymptotically optimal cost in the worst case but also perform well in other cases. Our advantage attributes to the rank space mapping before packing the data points. Such a packing strategy effectively incorporates the designs of both HR and STR, which both perform well on non-extreme data.

We also notice that HRR outperforms ZR. This suggests that when packing data points in the same rank space, the Hilbert curve yields a better packed R-tree than the Z-curve does. This result is consistent with an earlier study [33] that compares the query performance of R-trees packed with the Hilbert curve and the Z-curve in the same Euclidean space.

Regarding the baseline techniques, while TGS has smaller query I/O costs than our HRR and ZR on Skew and Tiger-East data (e.g., 3.29 vs. 3.95 and 5.82 when the query window area is 0.0001% of the data space on Tiger-East data, Fig. 10e), it is not worst-case optimal. Its query I/O costs are much higher than ours on the other data sets (e.g., 174.88 vs. 9.87 and 16.05 when the query window area is 0.0001% of the data space on Gaussian data, Fig. 10b). The other heuristic techniques HGO, HR, and STR share a similar limitation. In particular, the two Hilbert curve based techniques HGO and HR suffer the most on Cluster data (Fig. 10c), which is designed to test the worst-case performance of Hilbert R-trees. PR is the only baseline with worst-case optimal window query costs. Its empirical query costs, however, are consistently higher than those of HRR (e.g., 9.49 vs. 3.95 and 5.82 when the query window area is 0.0001% of the data space on Tiger-East data, Fig. 10e) and only slightly smaller than those of ZR on Skew data (Fig. 10d). For fairness, HGO and PR are designed for rectangles. They may not be optimal on point data for which HRR and ZR are designed.

To help further understand the benefit of the proposed packing strategy, we list the average output size (k/B) per query for Cluster data as follows. For the different query window areas tested (i.e., from 0.0001% to 2% of the data space size), the output sizes are 0.99, 0.99, 10.80, 98.73, 974.64, and 1936.29, respectively. Based on these output sizes and the relative query I/O costs shown in Fig. 10c, we can derive the absolute query I/O costs of the different R-trees. For example, at query

window area being 2%, the relative query I/O costs of HRR, ZR, TGS, PR, STR, HGO, and HR are 1.25, 1.28, 1.46, 1.59, 1.61, 50.40, and 51.37, which correspond to 2,420.36 (1,936.29×1.25), 2,478.45 (1,936.29×1.28), 2,826.98 (1,936.29×1.46), 3,078.70 (1,936.29×1.59) I/Os, 3,117.43 (1,936.29×1.61), 97,589.02 (1,936.29×50.40), and 99,467.22 (1,936.29×51.37), respectively. This means that HRR has at least 406.64 (14%) and up to 97,046.86 (98%) fewer I/Os than the baselines techniques. Similarly, ZR has at least 348.53 (12%) and up to 96,988.77 (98%) fewer I/Os than the baselines techniques. Note that these are improvements per query. For target applications such as digital mapping, there can be millions of user queries to be processed at the same time. The accumulated benefit of HRR and ZR over such a large number of queries is non-trivial.

Note that, for extremely small window queries (e.g., a point query), our techniques would be disadvantaged, because the R-tree query costs may be too small to justify the extra costs to access the B-trees for mapping the query window into the rank space. Under such a scenario, the baseline techniques may be preferred since they do not have the extra mapping costs.

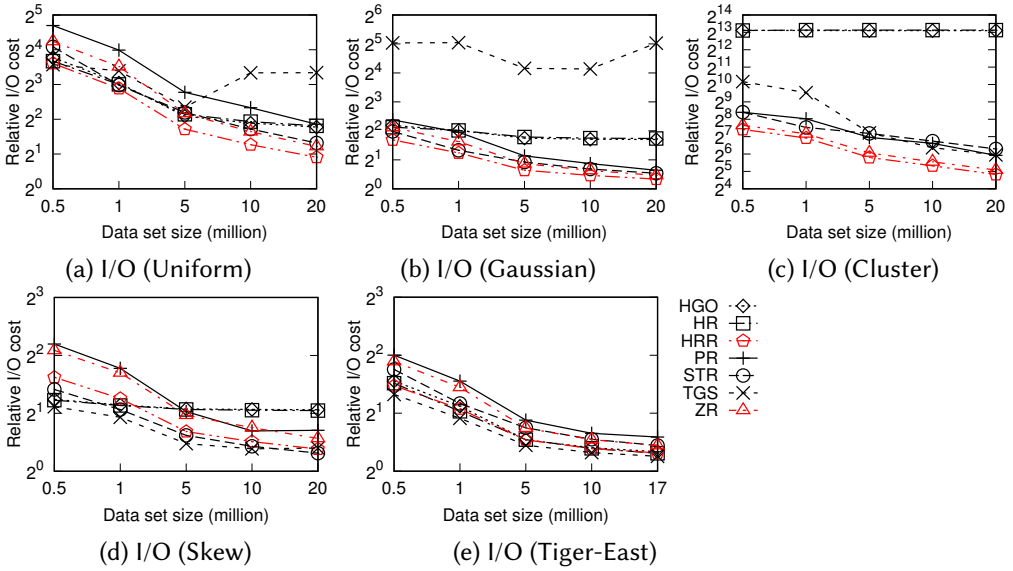


Fig. 11. Query costs – varying the data set cardinality

Varying the data set cardinality. Next, we vary the data set cardinality n from 0.5 to 20 million while keeping the query window size at 0.01% of the data space. For Tiger-East, we vary n to up to 17 million (i.e., the full Tiger-East data set) and generate the subsets by random sampling.

We see from Fig. 11 that the relative I/O costs drop as n increases for most techniques. This is because the data density increases with n. A query window overlapping a tree node may overlap more data points in this node, which causes the relative query I/Os to drop. TGS is an exception, and its query performance fluctuates. This technique relies on heuristics to minimize the area of the MBRs for the points packed together, which may not be optimal for all cases.

Our HRR technique again yields the smallest query I/O costs on Uniform, Gaussian, and Cluster data sets. On Tiger-East and Skew data, TGS has the smallest query I/O costs while those of HRR are close, e.g., 2.50 vs. 2.78 on 0.5 million Tiger-East data (Fig. 11e). As mentioned above, the query costs of TGS fluctuate, which can be over 25 times as large as those of HRR, e.g., 32.61 vs. 1.26 on 20 million Gaussian data (Fig. 11b). Our ZR technique has higher query costs than our HRR technique because of the different space-filling curves used, but it still preserves a consistently low query cost across the different data sets due to our rank space mapping based indexing technique.

STR has close query performance to that of HRR and ZR, for that these techniques share a similar design. However, on Cluster data, the performance difference is still non-trivial, e.g., 78.28 vs. 28.21 and 33.87 on 20 million data points (Fig. 11c). HGO and HR again suffer the most on Cluster data. Additionally, we see that the query costs of HGO and HR do not drop as fast as the other techniques on Gaussian, Cluster, and Skew (Figs. 11b, 11c, and 11d), where the data distributions are skewed. This can be explained as follows. HGO and HR use a grid of a fixed size to partition the data space. As n increases, there may be multiple points falling into the same grid cell, which are given the same Hilbert value, especially in the highly dense regions of the skewed data sets. This could lead to an arbitrary ordering for such points and impinge the performance of the resultant R-trees.

PR is again consistently outperformed by HRR and ZR, except for on 10 million Skew data where PR outperforms ZR slightly (Fig. 11d). On real data, our HRR and ZR reduce the query costs by up to 31% and 7%, respectively (2.78 and 3.71 vs. 4.00 on 0.5 million Tiger-East data, Fig. 11e). On synthetic data, our HRR and ZR reduce the query costs by up to 53% and 44%, respectively (28.21 and 33.87 vs. 60.53 on 20 million Cluster data, Fig. 11c).

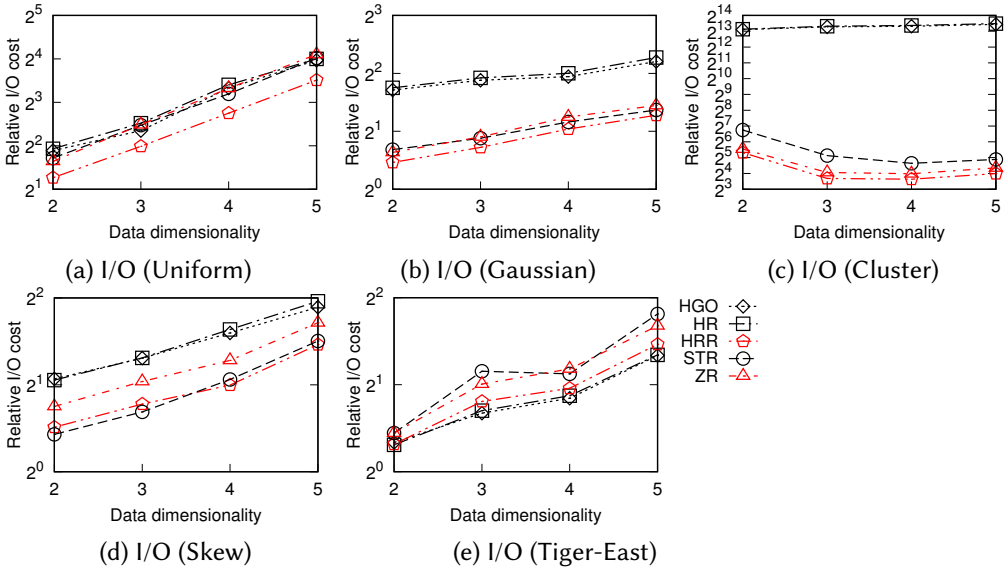


Fig. 12. Query costs – varying the data dimensionality

Varying the data dimensionality. We further vary the data dimensionality d from 2 to 5. The higher dimensional (i.e., $d > 2$) data sets are generated as follows. For Uniform and Gaussian, the coordinates of the data points in each dimension follow uniform and Gaussian ($\mu = 0.5$ and $\sigma = 1$) distributions, respectively. For Skew, the coordinates of the data points from a Uniform data set are raised to the power of $\alpha = 9$ for every dimension other than the first dimension. For Cluster, the data points are generated to form 10,000 clusters with centers evenly distributed on a horizontal line in the first dimension. Each cluster contains a subset of points following a uniform distribution in a 0.00001^d hypercube around the center. For Tiger-East, we add higher dimensional coordinates to the real data points via randomly picking coordinates from the first two dimensions.

We keep the query window size at 0.01% of the data space. On the Uniform, Gaussian, Skew, and Tiger-East data, the queries follow the data distribution and have a hypercube shape. On the Cluster data, a query window is a hyperrectangle where the edge in the first dimension spans across the data space (i.e., with length 1), while the edge in any other dimension is randomly placed and with length $(0.01\% \times \text{data space size})^{1/(d-1)}$.

Note that TGS and PR are dropped from the baselines for this set of experiments as their implementations [7] are hard-coded for $d = 2$.

As shown in Fig. 12, when d increases from 2 and 5, our HRR technique again outperforms the baselines on Uniform, Gaussian, and Cluster data while being close to the best baseline on Skew and Tiger-East data. Our ZR technique is also robust to the increase in d across the data sets.

An overall observation is that the relative query costs increase as d increases. This is expected because the data become more sparse as d increases, which leads to larger MBRs that may overlap with a query window but contribute few query answer points. There is an exception on Cluster data, where the relative query costs of HRR, ZR, and STR drop from $d = 2$ to $d = 4$ before rising again at $d = 5$ (Fig. 12c). We conjecture that this is because the Cluster data set occupies a data space of $[0, 1] \times [0, 0.00001]^{d-1}$ such that the projected distribution in the first dimension is clustered and the projected distribution in each of the other dimensions is uniform. Increasing d from 2 to 3 adds a dimension with a projected uniform distribution. This not only makes the overall data distribution more sparse but also makes it more uniform, which brings down the query costs (i.e., a less skewed data distribution tends to have lower query costs, cf. Figs. 12a and 12c). As d increases further, the impact of a more sparse data distribution becomes more prominent, and the relative query costs rise again. Note that the Hilbert curve based techniques HGO and HR do not benefit much from increasing d because the Cluster data set is designed to show their worst-case performance.

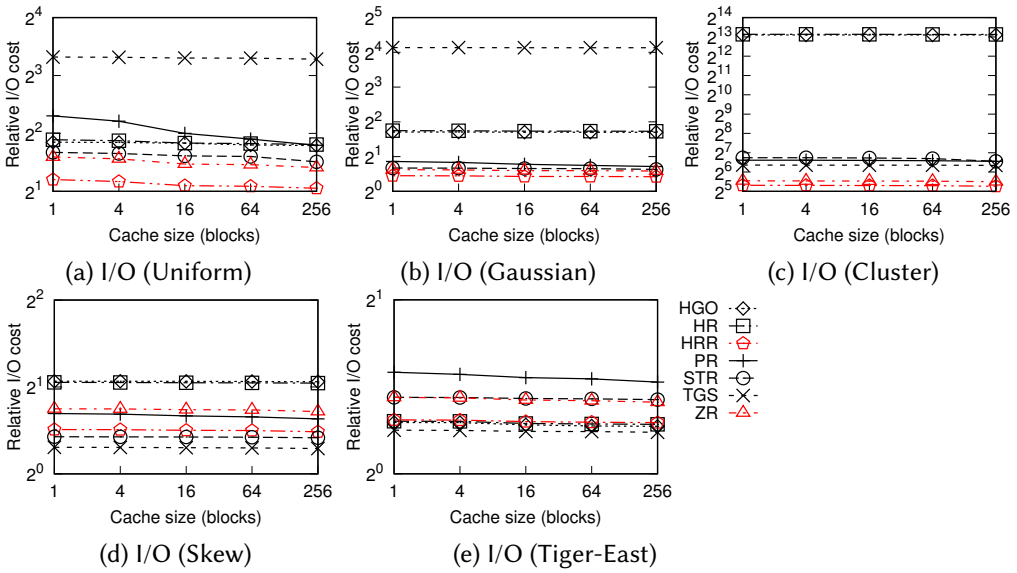
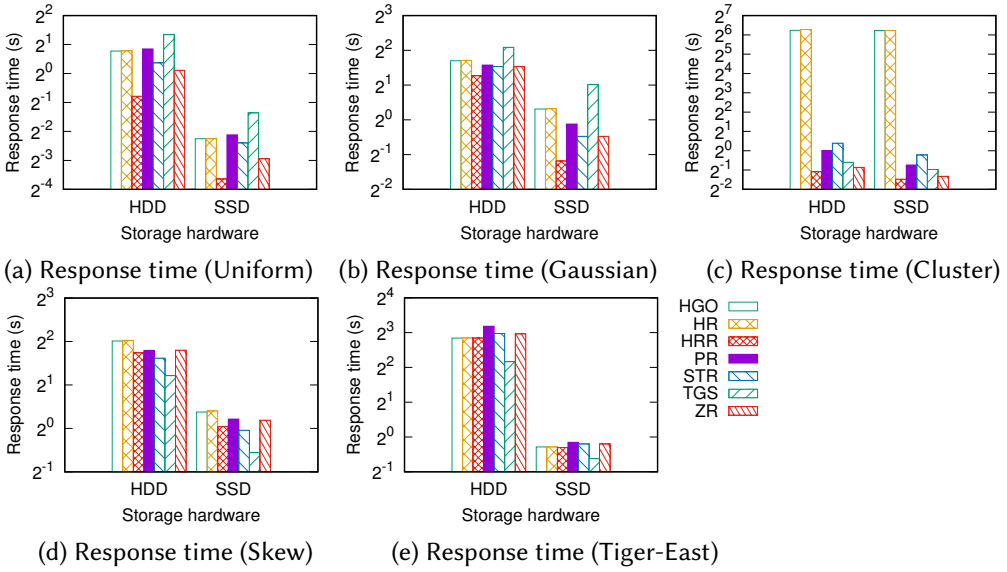


Fig. 13. Query costs – varying the cache size

Varying the cache size. In this set of experiments, we examine the impact of caching the tree nodes in main memory. We start caching from the root node to nodes in the lower levels of a tree. We vary the number of nodes cached from 1 to 256. From Fig. 13, we see that caching does not have a significant impact on the relative performance of the R-trees bulk-loaded by the different techniques. Our HRR and ZR techniques still obtain the best performance on Uniform, Gaussian, and Cluster data while they are also competitive on Skew and Tiger-East data. Note that our HRR and ZR techniques require two B-trees each for window query mapping. We cache the same number of nodes for each B-tree as that for the R-tree, i.e., our techniques require extra caching costs. However, this is just a constant time (i.e., 2 times) extra cost, which is worth paying to obtain the worst-case performance guarantee.

Overall, as the cache size increases, the query costs decrease for all techniques. The decrease in the costs may not seem too significant. This is because the main I/O costs come from accessing the leaf nodes, while even a cache with 256 blocks have not reached the leaf nodes yet (recall that our default data set size is 10 million, which means over 900 nodes at the parent level of the leaf nodes). For example, when the cache size increases from 1 (caching only the tree root) to 4 (caching the tree root and three child nodes of the root), the absolute query cost may drop by up to 3 I/Os. This means a drop of only 3% of the I/O costs, e.g., for TGS on Uniform data (cf. Fig. 13a), which has 92.72 I/Os per query on average. Also, not all cached nodes are accessed for every query. Thus, as the cache size increases further, the decrease in the I/O costs does not increase linearly with it.



(a) Response time (Uniform) (b) Response time (Gaussian) (c) Response time (Cluster)

(d) Response time (Skew) (e) Response time (Tiger-East)

Fig. 14. Query costs – impact of storage hardware

Impact of data storage hardware. To examine the impact of data storage hardware, we compare the query times using a solid-state drive (SSD) for index storage with those using a hard disk drive (HDD). We report the results on default data set size and query window size in Fig. 14.

All the techniques run faster on SSD. The speed-up can be up to 10 times, e.g., for PR on Tiger-East data (cf. Fig. 14e). Different techniques may have a different speed-up on different data sets, because they may form different data grouping. When the tree nodes accessed for query processing tend to be stored consecutively in the HDD, the speed-up offered by the SSD may be less significant. Thus, the performance gaps between different techniques may vary on HDD and SSD. For example, the performance gaps between our HRR and the baseline techniques are larger on the SSD than on the HDD on Gaussian data (cf. Fig. 14b). However, a faster technique on the HDD is also faster on the SSD in general. These observations confirm the adaptability of our techniques to SSDs.

***K*NN query processing.** While our structures are designed for window queries, they can also be adapted for *k*NN queries. We present *k*NN query performance results in an online appendix.

6.2.2 Bulk-loading. We implemented both the standalone bulk-loading algorithm (Section 3.2) and the parallel bulk-loading algorithm (Section 4.2). For the standalone algorithm (denoted by "**ZR**"), we measure the I/O, the response time on both HDD and SSD, and the index size. For the parallel algorithm, we measure the response time (denoted by "**ZR-R**"), the running time $\mathcal{T}$ (denoted by "**ZR-M**"), and the communication time (denoted by "**ZR-C**"), as described in Section 6.1.

We also implemented the bulk-loading algorithm for the proposed packing strategy using the Hilbert curve. We denote the standalone implementation by "**HRR**". As Figs. 15 and 16 show, HRR and ZR have very similar bulk-loading I/O and time costs. This is expected as they only differ in the curve used. Similar observation is made on the parallel implementation of the algorithms. To keep the figures concise, we omit the parallel HRR algorithm.

We report the results on Uniform and Tiger data in this subsection. Results on the other data sets show similar relative algorithm performance patterns and are omitted due to space limit. The similar relative algorithm performance across different data sets is expected. This is because the bulk-loading algorithms rely on sorting the data points. Different data sets may have an impact on the sorting efficiency, but such an impact is the same across the different bulk-loading algorithms since the same sorting algorithm is used (i.e., external merge sort in the TPIE library).

Bulk-loading on a single machine. We first show the algorithm performance when the algorithms are running on a single machine.

Varying the data set cardinality. We vary the data set cardinality and show the bulk-loading costs in Fig. 15 for Uniform and Tiger-East data. Here, we randomly sample the Tiger-East data set to obtain subsets of of different sizes.

We see from Figs. 15a and 15e that the bulk-loading I/O costs increase with the data set cardinality as expected. Both HR and STR outperform HRR and ZR in I/O cost, because they require fewer rounds of sorting. HR only sorts on the Hilbert-order values, while STR only sorts on the coordinates. Our HRR and ZR algorithms pay extra sorting costs in bulk-loading to achieve lower (and worst-case optimal) query costs, as shown above. PR has a slightly smaller I/O cost than those of HRR and ZR at start, but its I/O cost increases faster and gets very close to those of HRR and ZR when the data set cardinality exceeds 10 million. This can be explained by that PR needs to construct a pseudo-PR-tree for bulk-loading each level of the target R-tree. As there are more data points, the pseudo-PR-tree becomes taller and takes more I/Os to construct. TGS has a higher bulk-loading I/O cost than HRR and ZR due to its repetitive data access for optimization function computation. HGO has the highest bulk-loading I/O cost, which is incurred by accessing the data points to compute the $gopt^*$ cost values: for each $gopt^*(i), i \in [b-1, n-1]$, a block of data points is needed for MBR area computation. Note that, in our HGO implementation, we only buffer a block of B $gopt^*$ cost values but not the data points. This is to keep in line with the rest of the algorithms which are all based on external memory (including sorting which is done by external merge sort). When more memory are available to buffer data points, the I/O cost of HGO is expected to be lower.

The bulk-loading times are shown in Figs. 15b, 15d, 15f, and 15h. We see that the comparative performance of the algorithms in terms of response time is consistent with that in the I/O cost. HR and STR have the lowest response times for their least amount of sorting workload. HRR, ZR, and PR have close response times which are higher than those of HR and STR, since they need to do more sorting. TGS and HGO have the highest response times due to their higher I/O costs for cost function computation. Comparing the HDD times with the SSD times on the same data set (e.g., Figs. 15b and 15d), we see that the algorithms run (up to 2.5 times) faster on SSD, although now the advantage of SSD is less significant than that in query processing. This is because (1) computation in the bulk-loading process (e.g., sorting) takes a more significant portion of the overall response time than computation in querying processing; and (2) bulk-loading requires more sequential accesses, i.e., scanning and (merge) sorting the data points, where the performance of HDD suffers less.

The bulk-loaded index sizes are shown in Figs. 15c and 15g. The baselines HR, PR, STR, and TGS all have the same index size, as they all pack every B points into a leaf node of the R-trees. Our HRR and ZR techniques create d B-tree indices in addition to an R-tree bulk-loaded. This results in about 78% larger index sizes. For HGO, we set $b = B/3$ following its original proposal [1] such that a leaf node of an R-tree may have at least $B/3$ and at most B data points. We observe that the

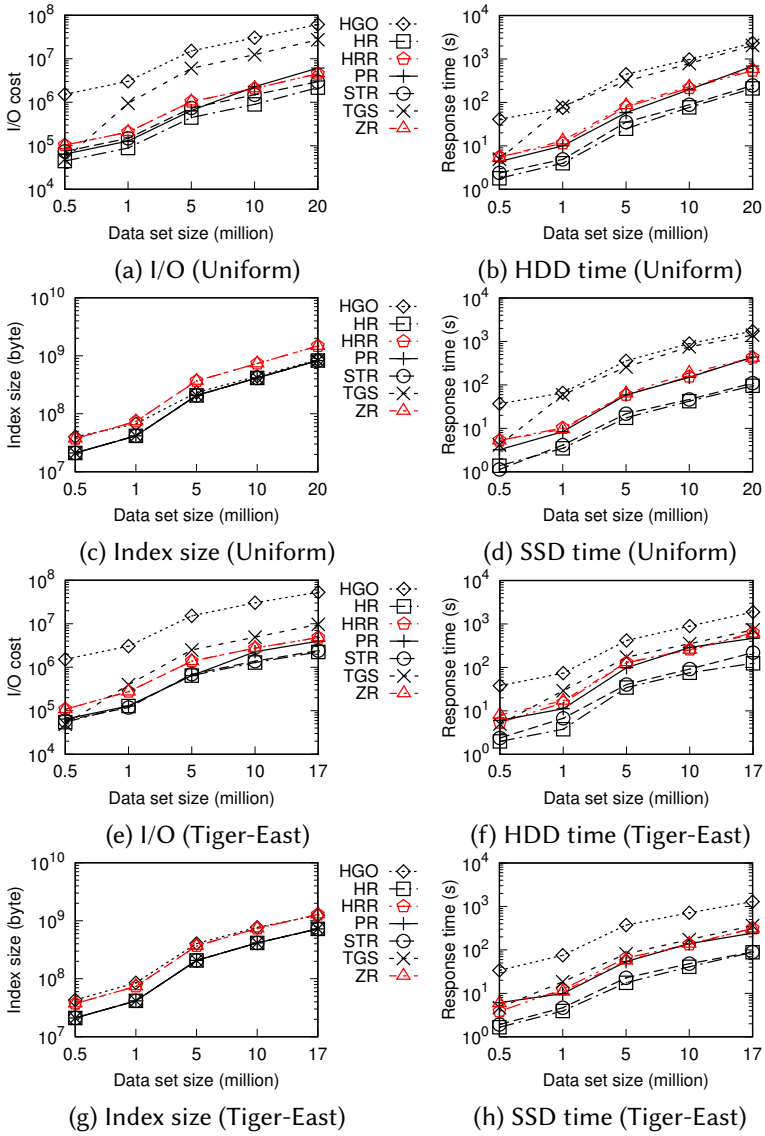


Fig. 15. Bulk-loading costs – standalone algorithms (varying the data set cardinality)

R-trees created by HGO can be up to twice as large (Fig. 15g) as those created by HR, PR, STR, and TGS. This means a 50% storage utilization of the leaf nodes, which is lower than the 80% storage utilization reported in the original proposal. The lower storage utilization can be explained by that we use point data and assume unknown query profile by setting the query size to be zero. Under such settings, it makes sense to group small numbers of close points together (i.e., leaving as little blank space in an MBR as possible), so as to minimize the MBR areas. As a result, more groups of points (i.e., leaf nodes) are created, which leads to the lower storage utilization. In contrast, rectangular data are used in the original proposal. The height and the width of the data rectangles are further expanded by a query window size, after which many rectangles may be overlapping already. In this case, it makes sense to group more overlapping rectangles together, such that the

overlapping area only contributes to the total MBR area once. Thus, a higher storage utilization was obtained. This also explains why the curve of HGO becomes closer to those of HR, PR, STR, and TGS as there are more data points (Figs. 15c and 15g), since denser data allow grouping more points together without including much blank space in the resultant MBRs.

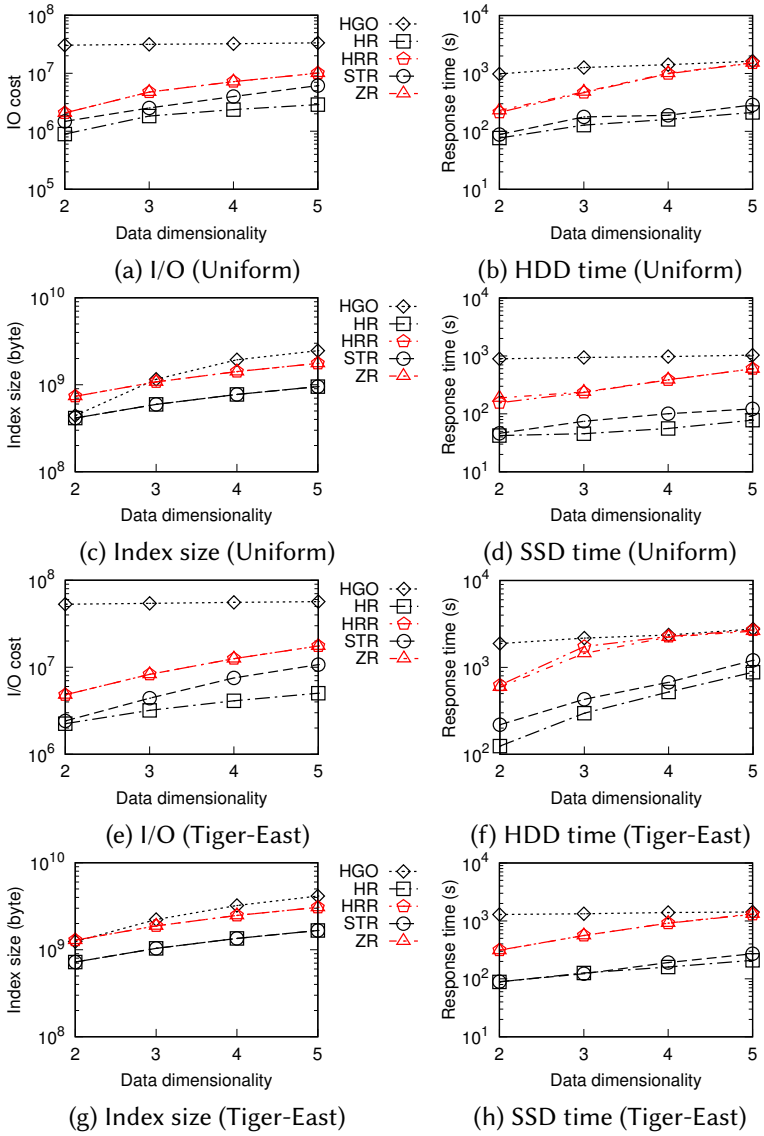


Fig. 16. Bulk-loading costs – standalone algorithms (varying the data dimensionality)

Varying the data dimensionality. We further vary the data dimensionality from 2 to 5. As shown in Fig. 16, the bulk-loading costs increase as d increases, since the size of each data point and the number of data blocks to be processed increase with d. The relative performance of the algorithms is similar to that when the data set cardinality is varied, i.e., HR and STR have the lowest bulk-loading costs; our HRR and ZR have higher costs than HR and STR for our more sorting rounds; and HGO has the highest costs for its cost value computations. We note that the increase in the bulk-loading

costs of HGO is the slowest. This is because its cost value computation takes a data block access for each $gopt^*(i), i \in [b-1, n-1]$, which is not impacted by d.

Focusing on our own techniques HRR and ZR, we see that our bulk-loading costs and the resultant index sizes do not increase drastically with d. This is because our additional computation (and I/O) costs are just for d rounds of sorting, and our additional storage space requirement is for d B-trees, both of which scale linearly with d.

Parallel bulk-loading. Next, we study the performance of our parallel bulk-loading algorithm.

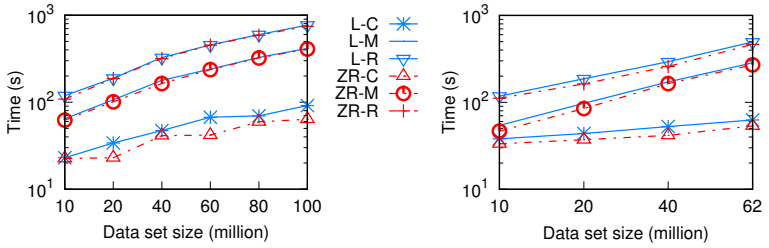


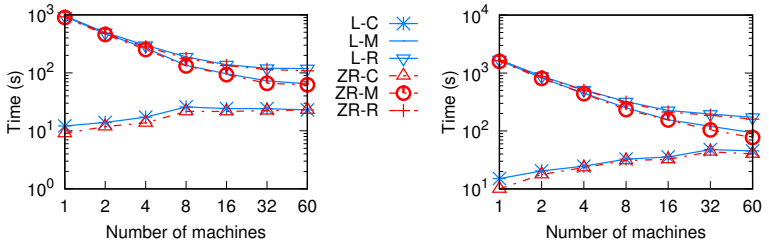
Fig. 17. Bulk-loading costs – parallel algorithms (varying the data set cardinality)

Varying the data set cardinality. We scale our experiments to 100 million points of synthetic data and 62 million points of real data (i.e., Tiger-Full). Subsets of real data are generated by randomly sampling the Tiger-Full data set. We show in Fig. 17 the communication time (ZR-C), running time (ZR-M), and response time (ZR-R) of the proposed parallel bulk-loading algorithm. We observe that ZR-C, ZR-M, and ZR-R are consistently smaller than their level-by-level counterparts L-C, L-M, and L-R. ZR-C is up to 38% smaller than L-C (at 60 million Uniform data) due to the smaller number of communication rounds of the proposed algorithm, while ZR-M is only up to 14% smaller than L-M (at 10 million Tiger-Full data) since both algorithms perform similar computations. Overall, the response time ZR-R is up to 13% smaller than L-R (at 20 million Tiger-Full data). Note that the response time includes the time to write the bulk-loaded R-tree back to a single machine for query processing. This writing requires a large number of I/Os on a single machine, which makes up for about two thirds of the response time and is the same for both algorithms. The benefit of the proposed algorithm would be more significant if this writing time is left out. Also, the improvements are obtained over R-trees with relatively low heights (e.g., 4 for 100 million data points), where the execution of the proposed parallel algorithm and the level-by-level parallel algorithm differs by no more than two rounds. When the tree height gets larger and there are more rounds, the performance improvement is expected to be higher.

Meanwhile, by comparing Figs. 15b and 17a, we see that, on 10 million data points, both the response time (ZR-R, 108.61 seconds) and the running time (ZR-M, 62.20 seconds) of the proposed parallel algorithm are smaller than the running time of the standalone implementation ZR (229.18 seconds) and the baseline algorithm PR (196.81 seconds). The advantage of the running time ZR-M over PR is 68%, and this advantage grows with the data set size (e.g., 85% on 20 million data points), demonstrating the scalability of the proposed parallel algorithm.

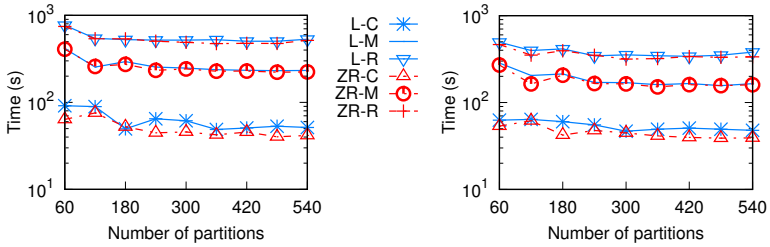
Varying the number of participating machines g. We further examine the scalability of our parallel bulk-loading algorithm by varying the number of participating machines (i.e., worker machines) g in the cluster from 1 to 60 (which is the number of all worker machines in our cluster). To suit the capacity of a single worker machine, we use 10 million synthetic points and 17 million real data points (i.e., Tiger-East data) in this set of experiments. As shown in Fig. 18, when the number of worker machines increases, the response times (Z-R and L-R) and the running times (Z-M and L-M) decrease while the communication times (Z-C and L-C) increase. These observations are

expected. Sharing the workload by more machines shortens the overall response time as well as the running time of each machine, but it also creates more communication costs to transfer the workload. Our algorithm costs Z-R, Z-M, and Z-C again outperform their level-by-level counterparts consistently. Another observation is that our response time Z-R decreases almost linearly as g increases from 1 to 8, which confirms the strong scalability of our proposed parallel bulk-loading algorithm. When g increases further, the decrease in Z-R slows down, as there are higher scheduling and communication costs for running more machines in the cluster.



(a) Time (Uniform) (b) Time (Tiger-East)

Fig. 18. Bulk-loading costs – parallel algorithms (varying the number of participating machines g)



(a) Time (Uniform) (b) Time (Tiger-Full)

Fig. 19. Bulk-loading costs – parallel algorithms (varying the number of partitions)

Varying the number of partitions. Spark allows creating more partitions than the worker machines available, such that faster machines can be allocated with more partitions while slower machines can be allocated with fewer partitions (i.e., to achieve a better load balancing). In this set of experiments, we study how the number of partitions impacts our algorithm performance. In Fig. 19, we show the results where the number of partitions is varied from 60 to 540 for 100 million synthetic data points and 62 million real data points (i.e., Tiger-Full data). We see that, increasing the number of partitions helps reduce the algorithm times initially. The most significant drop in the times is observed when the number of partitions increases from 60 to 120 (e.g., Z-R is reduced by 28% on 100 million Uniform data). This is consistent with the Spark Programming Guide [56] which suggests to set the number of partitions to be 2 to 4 times of the number of worker machines g (i.e., 120 to 240 since we have $g = 60$). When the number of partitions increases further, the benefit fades away because more scheduling costs are incurred. There are fluctuations in the algorithms times because of the unstableness of the virtual nodes on which the algorithms are run. When the number of partitions reaches 540, a more obvious increase in the response times Z-R and L-R is observed.

6.2.3 Update Handling. This subsection evaluates the performance of the dynamic structures. We first bulk-load LogR-trees (LogR-H and LogR-Z), LogR*-trees (LogR*-H and LogR*-Z), and the LR-tree with the same set of one million data points. Then, we insert points into or delete points from the trees. We run 100 window queries over the trees after all updates are completed. The

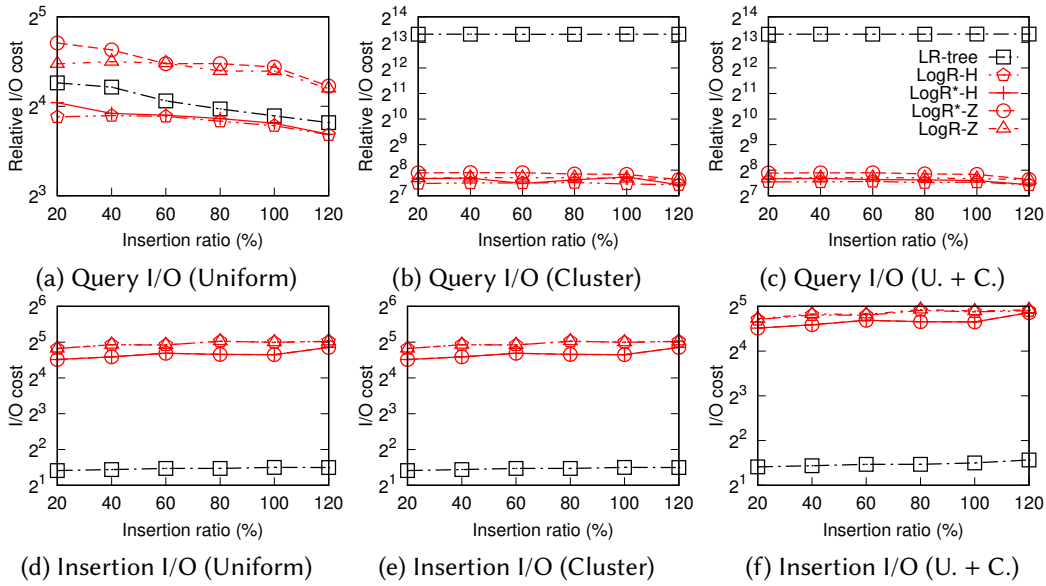


Fig. 20. Query and update I/O costs – insertion (Uniform and Cluster data)

window queries are generated in the same way as described in Section 6.2.1. The relative I/O cost per window query and the average I/O cost per update are measured and reported.

Insertion. We first study the impact of insertions by inserting from 20% to 120% new data points (i.e., 200,000 to 1,200,000 insertions) into the dynamic structures (after the initial trees have been bulk-loaded). The data points inserted follow the same distributions as those of the initial data sets. In particular, for the experiments on Tiger-East data, the initial data points and the inserted data points are disjoint random samples of the original Tiger-East data set. An exception is the "Uniform+Cluster" set of experiments (denoted by "U. + C." in Figs. 20c and 20f), where the initial points follow a uniform distribution while the inserted points follow a clustered distribution (same as the Cluster data). This set of experiments aims to test the capability of our proposed dynamic structures to preserve the worst-case query cost optimality when the data distribution changes.

From Fig. 20, we see that, as more points are inserted, the relative query I/O costs decrease overall, and there are fluctuations (e.g., Fig. 20a). The decreasing trend is because, when the points become more dense, a query window overlapping with a tree node has a higher probability to overlap with the data points in this node, i.e., there are lower percentages of extra query I/Os that do not contribute any output. The cost fluctuations are caused by the periodic bulk-loading of a series of R-trees for hosting the inserted points. As more points are inserted, more R-trees are created. Among these R-trees, the smaller ones may cause higher relative query I/Os, as the points in their nodes are more sparse. This impinges the overall decreasing trend of the query I/O costs. Note that the number of R-trees does not increase constantly. When a series of R-trees are all full, they are destroyed and rebuilt into a single larger R-tree. At this moment, the relative query I/O costs decrease again. Also, since the LogR-trees, LogR*-trees, and LR-trees have different fanout (i.e., values of B), they may have different rebuilding and query cost fluctuation cycles.

The insertion I/O costs increase slowly as there are more points inserted, and there are also fluctuations (e.g., Fig. 20d). The slow increasing trend is expected as the amortized insertion I/O cost is in the scale of $\log_B n$ where B is quite large (i.e., $B \geq 73$). Even for the smallest B value 73, when n grows from the initial value $1,000,000$ by 120% to $2,200,000$, the cost difference is

only $\log_{73} 2,200,000 - \log_{73} 1,000,000 \approx 3.40 - 3.22 = 0.18$. The fluctuations in the insertion I/O costs can also be explained by the periodic bulk-loading of the R-trees. When a large R-tree is bulk-loaded, there will be a peak in the insertion I/O costs. Then, the insertion I/O costs may drop slightly until the next bulk-loading of another large R-tree.

Focusing on the comparison among the different structures, we see that, in terms of the query I/O costs, both proposed structures LogR-H and LogR*-H outperform the baseline structure LR-tree consistently across Uniform, Cluster, and Uniform+Cluster data (Figs. 20a, 20b, and 20c). In particular, on Cluster data, LogR-H and LogR*-H reduce the relative query I/O cost by up to 98%, e.g., 172.40 (LogR-H) vs. 10,227.56 (LR-tree) for query processing after 120% insertions. Similar query I/O cost reduction is observed on Uniform+Cluster data, which confirms the capability of our LogR-trees and LogR*-trees to preserve the worst-case query I/O cost optimality. Between our structures LogR-H and LogR*-H, LogR*-H yields slightly higher query I/O costs. This is because LogR*-H stores extra dictionary pointers in its R-trees to improve insertion performance, which leads to a smaller fanout and hence more I/Os to fetch the query answer. LogR-Z and LogR*-Z have higher query I/O costs than LogR-H and LogR*-H (and LR-tree on Uniform data) do. They use the Z-curve which is known to be outperformed by the Hilbert-curve used by the other trees.

The lower query I/O costs of our structures come with higher insertion costs (Figs. 20d, 20e, and 20f). The extra insertion costs are incurred by the rank space mapping and the ID B-tree maintenance, which are not required by LR-tree. We argue that the extra insertion costs (e.g., 32.45 vs. 2.82 for 120% insertions of LogR-H and LR-tree on Cluster data) are worth spending, considering the significant gains in the query performance. Also, using our proposed parallel bulk-loading strategy, we can further bring down the bulk-loading times of the R-trees in our structures. Another observation is that LogR*-H and LogR*-Z have very similar insertion I/O costs, which is expected as they only differ in the curve value computation. They both have lower insertion I/O costs than those of LogR-H and LogR-Z, and the improvement is up to 23%, e.g., 25.24 vs. 32.59 after 80% insertions on Uniform data (Fig. 20d). On the same data set, the extra query I/O costs paid by LogR*-H comparing with LogR-H are just up to 12%, e.g., 16.45 vs. 14.73 after 20% insertions (Fig. 20a). This verifies the effectiveness of our insertion improvement technique for the LogR*-trees.

The experimental results on Gaussian, Skew, and Tiger-East data are shown in Fig. 21. The overall performance patterns of the different structures resemble those in Fig. 20. We notice that LR-tree has slightly lower query I/O costs than those of the proposed structures on Skew data, e.g., 2.58 vs. 3.28 for LR-tree and LogR-H after 100% insertions (Fig. 21b). This is because LogR*-trees and LogR-trees may have slightly more tree nodes than LR-trees due to their smaller B values, which cost more query I/Os. On Skew data, such extra costs dominate the query cost reduction achieved by our rank space based R-trees in LogR*-trees and LogR-trees over Hilbert R-trees in LR-tree.

Deletion. Next, we study the impact of deletions by deleting from 20% to 100% of the initial data points (i.e., 200,000 to 1,000,000 deletions). The points are randomly deleted, except for the "Uniform→Cluster" data (denoted by "U. → C." in Figs. 22c and 22f). For this set of experiments, the initial data points follow a uniform distribution, and 10% to 50% of the points are later deleted to form a clustered distribution, i.e., only points outside the clusters are deleted to form a data set similar to the Cluster data. The aim is to test the capability of our proposed dynamic structures to preserve the worst-case query cost optimality when the data distribution changes.

As shown in Fig. 22, when more points are deleted, the relative query I/O costs increase. This is because, as the points become more sparse, a query becomes more likely to overlap with a tree node but few points inside the node. When all the points (100%) have been deleted, there is no query cost. We denote the relative query I/O cost by 1 in this case to suit the logarithmic notation of the figures. LogR-H and LogR*-H again outperform LR-tree consistently across Uniform, Cluster, and Uniform→Cluster data (Figs. 22a, 22b, and 22c). On Cluster data, LogR-H and LogR*-H reduce

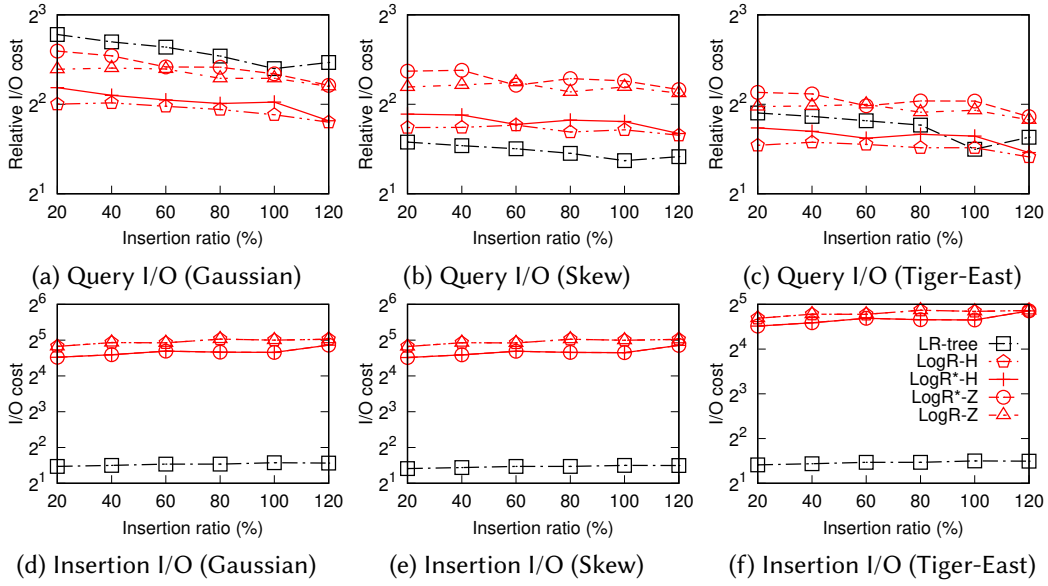


Fig. 21. Query and update I/O costs – insertion (Gaussian, Skew, and Tiger-East data)

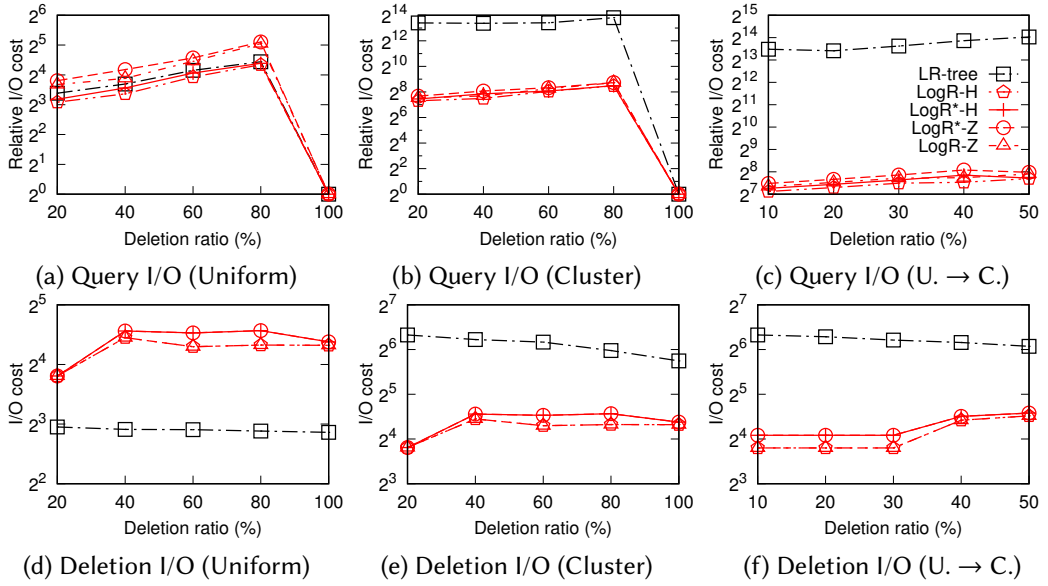


Fig. 22. Query and update I/O costs – deletion (Uniform and Cluster data)

the relative query I/O cost by up to 98%, e.g., 158.85 (LogR-H) vs. 10,832.98 (LR-tree) after 20% deletions. Similar cost reduction is observed on Uniform→Cluster data, which again confirms the capability of LogR-trees and LogR*-trees to preserve the worst-case query I/O cost optimality.

The deletion I/O costs of the LogR-trees and LogR*-trees increase between 20% and 40% deletions. This is because, as more points are deleted, the nodes in the ID B-trees underflow, which need to be merged and cause extra I/Os. After the merging, the R-tree nodes in LogR*-H and LogR*-Z need to be accessed to further update the dictionary pointers, which explains for their higher I/O costs

than those of LogR-H and LogR-Z. There is a drop in the deletion I/O costs of the LogR-trees and LogR*-trees between 40% and 60% deletions. This is because of an overhaul of these trees after 50% of the points are deleted, which creates more compact ID B-trees and reduces the node merging later on. In comparison, LR-tree simply lets the R-tree nodes underflow, and it does not have an ID B-tree. No node merging occur. Note also that, since LR-tree does not have an ID B-tree, it only supports deletions by data point coordinates but not by ids, which can be very expensive on highly skewed data, e.g., Cluster and Uniform→Cluster data (Figs. 22e and 22f). This explains for the higher deletion I/O costs of LR-tree than those of the proposed structures, noting that this may not be an exactly fair comparison since our structures use deletions by ids.

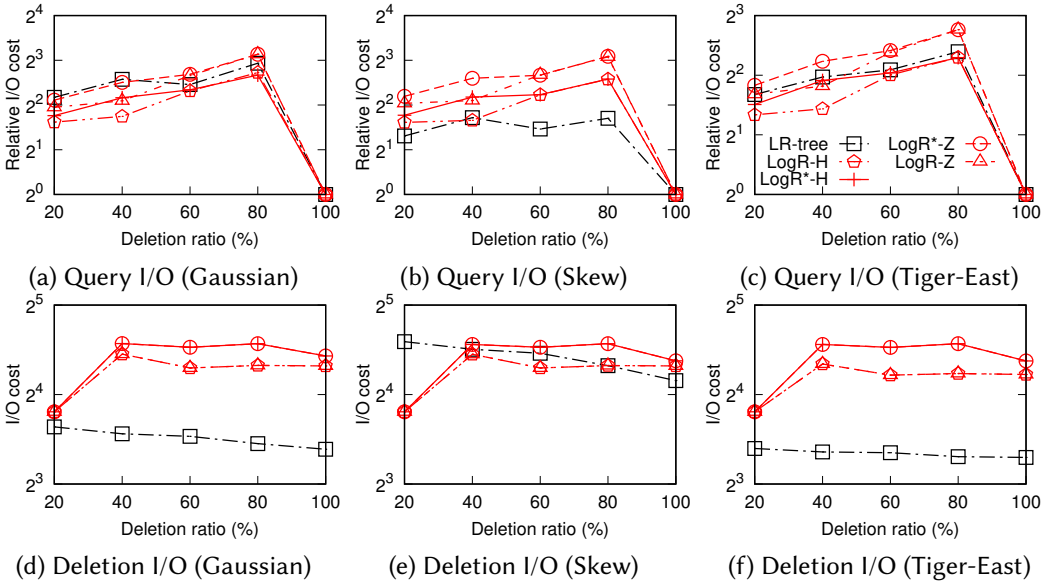


Fig. 23. Query and update I/O costs – deletion (Gaussian, Skew, and Tiger-East data)

The experimental results on Gaussian, Skew, and Tiger-East data are shown in Fig. 23. The overall comparative performance patterns again resemble those in Fig. 22, with an exception that LR-tree has slightly lower query I/O costs than those of the proposed structures on Skew data, e.g., 3.26 vs. 5.97 for LR-tree and LogR-H after 80% deletions (Fig. 23b).

Impact of caching. In this set of experiments, we further examine the impact of caching the tree nodes. For query processing, we follow the caching experiments in Section 6.2.1 and start caching from the root node of every R-tree in the logarithmic structure to nodes in the lower levels. For data insertion and deletion, we implement a cache using the *least recently used* (LRU) replacement strategy. We vary the number of nodes cached from 1 to 256.

We show in Fig. 24 the impact of caching on query and index update costs with data insertions. Here, we insert 80% more data points into the indices and then query the updated indices. As expected, when the cache size increases, the query and data update costs decrease for all techniques. Comparing with that on the caching experiments in Section 6.2.1 (Fig. 13), the query costs now decrease more significantly as the cache size increases (especially when the cache size exceeds 16 blocks, e.g., Figs. 24a to 24c). This is because, as mentioned above, the update experiments here are done on smaller data sets (with one million data points) for efficiency consideration. A smaller number of I/Os is needed for query processing on these smaller data sets. Thus, the I/Os saved by caching now takes up a larger portion of the overall I/O costs.

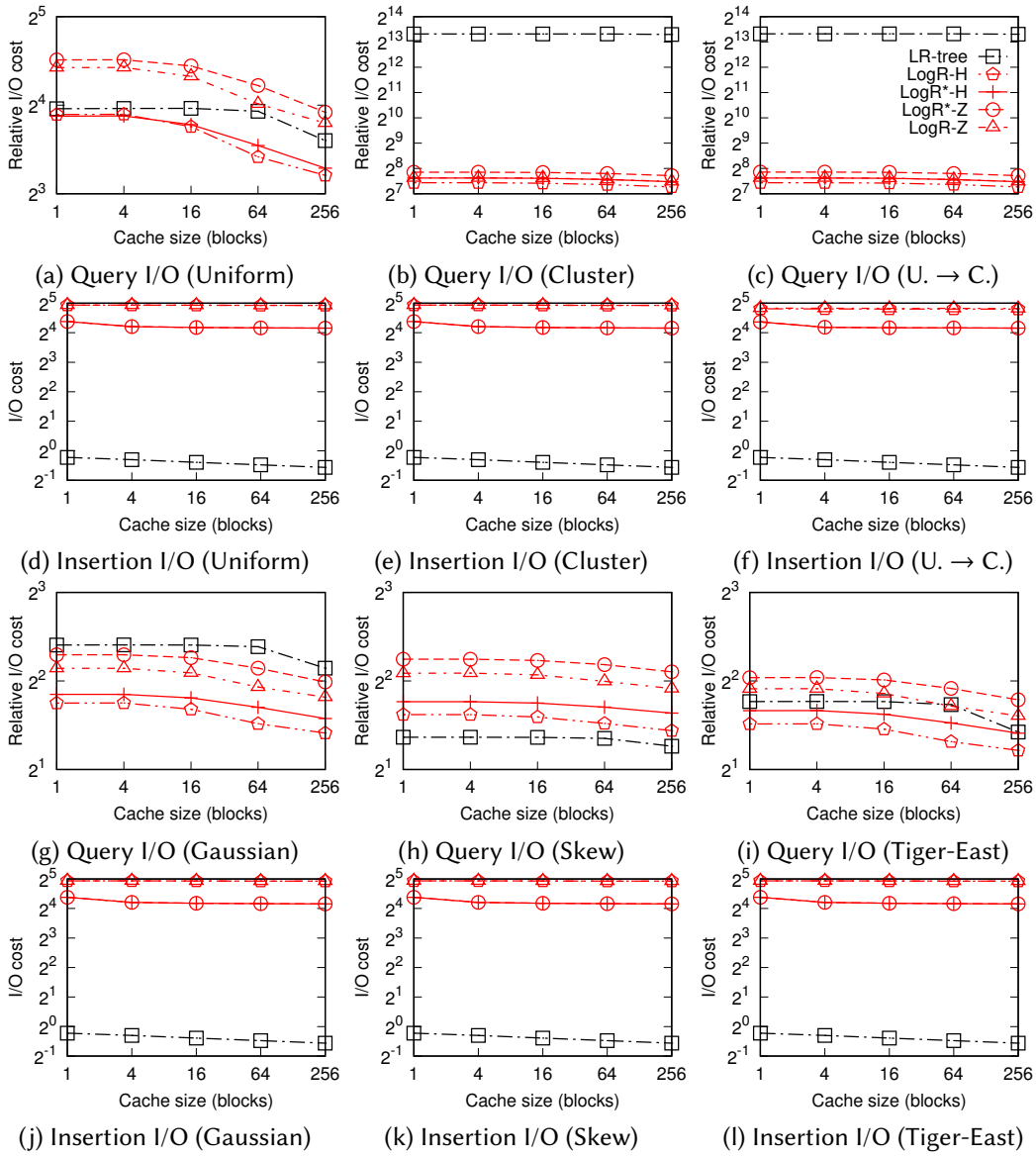


Fig. 24. Query and update I/O costs – impact of the cache size (insertion)

We also see that caching does not have a significant impact on the relative algorithm performance. Similar to what has been observed in the experiments without caching, our LogR*-H and LogR-H techniques still outperform the baseline technique LR-tree in the query costs on all data distributions except for Skew data. Note that our techniques require two B-trees for each R-tree in the logarithmic tree structure for window query mapping. Similar to the caching experiments in Section 6.2.1, we cache the same number of nodes for each B-tree as that for the R-tree, i.e., our techniques require a small constant time (i.e., 2 times) extra caching cost, to obtain the performance guarantee.

In terms of the index update (i.e., data insertion) costs, our techniques are still outperformed by the LR-tree after caching, as we need to maintain a more complex data structure. The performance

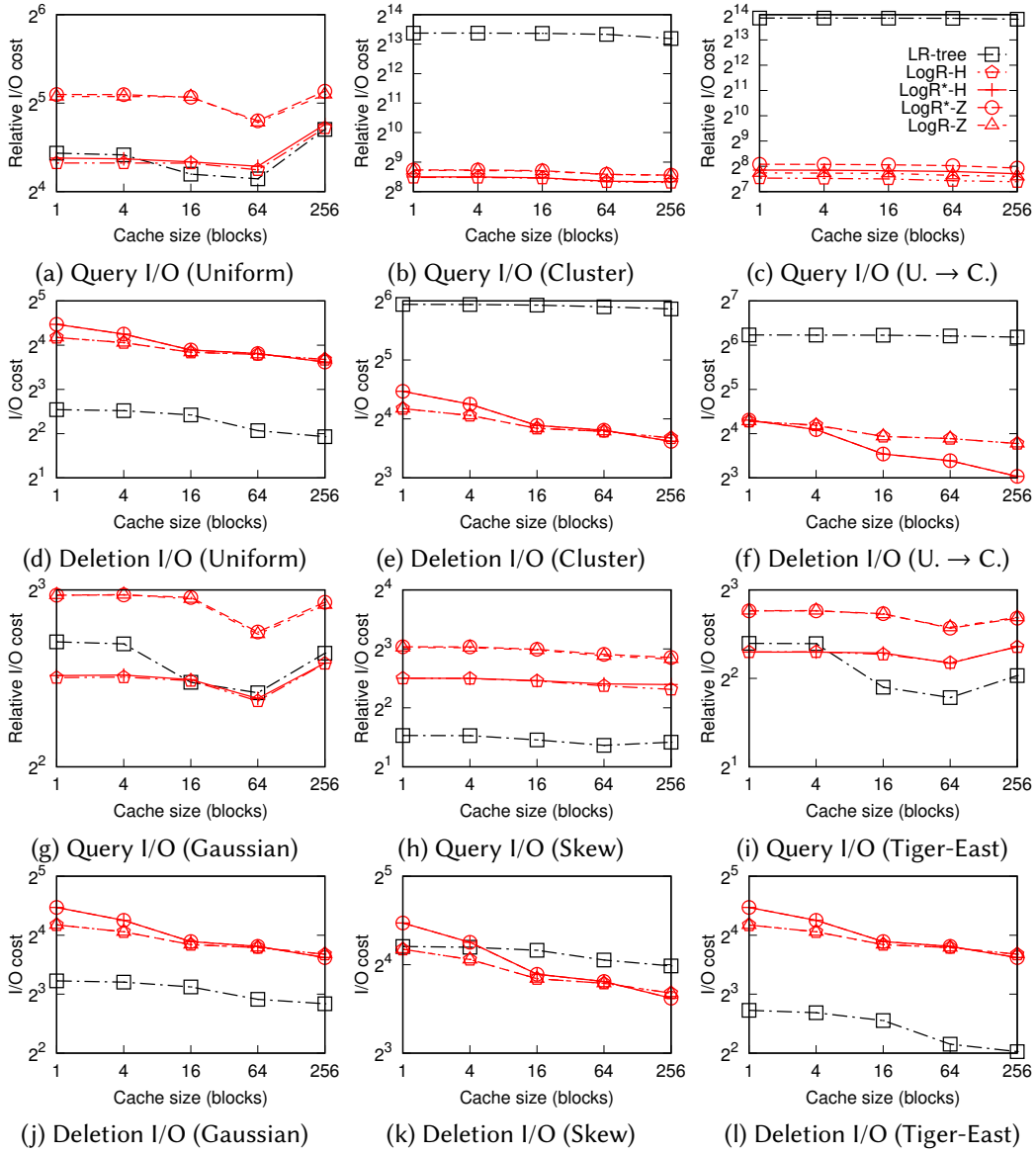


Fig. 25. Query and update I/O costs – impact of cache size (deletion)

gap does not increase with the cache size (note the logarithmic scale in the y-axis of the figures). This shows that our techniques can take advantage of the cache as well as the LR-tree does.

For data deletions, we show the impact of caching in Fig. 25, where we delete 80% (40% for the "Uniform→Cluster" data) of the data points and then query the updated indices. The overall observations are similar to that in Fig. 24. The query and index update costs drop when the cache size increases, while the cache dose not significantly impact the relative performance of the different techniques. There are two exceptions. First, the query costs rise back on the Uniform, Gaussian, and Tiger-East data when the cache size reaches 256 blocks (Figs. 25a and 25g). This is because we counted the cost of pre-loading the tree nodes into the cache as part of the query I/O costs. When

256 tree nodes are cached, the pre-loading I/O costs outweigh the reduction in the query costs (on Uniform, Gaussian, and Tiger-East data which are less skew and have lower query costs), which causes the rise in the overall query costs. Second, on Uniform and Tiger-East data, LR-tree now yields slightly lower query I/O costs when the cache size reaches 16 blocks, e.g., 3.44 vs. 4.50 for LR-tree and LogR-H when caching 64 blocks on Tiger-East data (Fig. 25i). However, LR-tree is still much worse than our techniques on worst-case workloads, e.g., on Cluster data (cf. Fig. 25b).

Insertion and deletion. Experiments where there are both insertions and deletions show consistent results to the above. We omit the results due to space limit.

7 CONCLUSIONS

We revisited a classic spatial index, the R-tree, and proposed an R-tree packing strategy to construct R-trees that are worst-case optimal and empirically efficient for query processing. This packing strategy maps data points into a rank space where the points are packed by their Z-order values. Mapping into a rank space avoids data points with the same coordinates. This overcomes the difficulty of space-filling curve based indices in offering optimal query performance in worst-case scenarios [7, 65]. It results in an R-tree structure that can answer a window query with $O((n/B)^{1-1/d} + k/B)$ I/Os in the worst case, which is asymptotically optimal. Experiments on both real and synthetic data confirmed the query efficiency of such an R-tree: on real data, the query I/O cost of the R-tree is up to 31% lower than that of PR-trees and similar to that of STR-trees; on highly skewed synthetic data, the query I/O cost of the R-tree is 54% lower than that of PR-trees and 64% lower than that of STR-trees. Another advantage of this packing strategy is that it only relies on sorting, which well suits parallel bulk-loading of R-trees over large data sets. We proposed a parallel R-tree bulk-loading algorithm based on this packing strategy using the MapReduce model. The algorithm takes only $O(\log_s n)$ rounds of computation to bulk-load an R-tree. It outperforms the PR-tree bulk-loading algorithm in running time by 85% on large data sets with 20 million data points. We also considered data update handling. Our R-tree based dynamic index structures can process data insertions and deletions without compromising the worst-case query I/O cost optimality. These proposed dynamic index structures achieve up to 98% lower query I/O costs comparing with the LR-tree – a Hilbert R-tree variant with update supports. The advantage is most significant when the data distribution is highly skewed.

For future work, we are interested in applying the rank space technique over other indices such as quad-trees and GiMP [63] to optimize window query processing. De-amortizing the update cost to avoid workload peaks for global rebuilds would be another interesting direction to explore.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

This work is supported in part by Australian Research Council (ARC) Discovery Project DP180103332, a direct grant (Project Number: 4055079) from The Chinese University of Hong Kong, and a Faculty Research Award from Google.

REFERENCES

[1] Daniar Achakeev, Bernhard Seeger, and Peter Widmayer. 2012. Sort-based Query-adaptive Loading of R-trees. In *CIKM*. 2080–2084.

[2] Daniar Achakeev, Marc Seidemann, Markus Schmidt, and Bernhard Seeger. 2012. Sort-based Parallel Loading of R-trees. In *1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. 62–70.

[3] Pankaj K. Agarwal, Lars Arge, Octavian Procopiuc, and Jeffrey Scott Vitter. 2001. A Framework for Index Bulk Loading and Dynamization. In *28th International Colloquium on Automata, Languages and Programming*. 115–127.

[4] Pankaj K. Agarwal, Mark de Berg, Joachim Gudmundsson, Mikael Hammar, and Herman J. Haverkort. 2001. Box-trees and R-trees with Near-optimal Query Time. In *17th Annual Symposium on Computational Geometry (SoCG)*. 124–133.

[5] Pankaj K. Agarwal, Kyle Fox, Kamesh Munagala, and Abhinandan Nath. 2016. Parallel Algorithms for Constructing Range and Nearest-Neighbor Searching Data Structures. In *PODS*. 429–440.

[6] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. 2014. Parallel Algorithms for Geometric Graph Problems. In *STOC*. 574–583.

[7] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. 2008. The Priority R-tree: A Practically Efficient and Worst-case Optimal R-tree. *ACM Transactions on Algorithms* 4, 1, Article 9 (2008), 9:1–9:30 pages.

[8] Lars Arge and Jan Vahrenhold. 2004. I/O-efficient Dynamic Planar Point Location. *Computational Geometry* 29, 2 (2004), 147 – 162.

[9] Lars Arge and Jeffrey Scott Vitter. 2003. Optimal External Memory Interval Management. *SIAM J. Comput.* 32, 6 (2003), 1488–1508.

[10] Paul Beame, Paraschos Koutris, and Dan Suciu. 2013. Communication Steps for Parallel Query Processing. In *PODS*. 273–284.

[11] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*. 322–331.

[12] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[13] Jon Louis Bentley. 1979. Decomposable Searching Problems. *Inform. Process. Lett.* 8, 5 (1979), 244 – 251.

[14] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree : An Index Structure for High-Dimensional Data. In *VLDB*. 28–39.

[15] Mark Berg, Marc Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. 2000. *Computational Geometry*. Springer Berlin Heidelberg.

[16] Panayiotis Bozanis, Alexandros Nanopoulos, and Yannis Manolopoulos. 2003. LR-tree: A Logarithmic Decomposable Spatial Index Method. *Computer Journal* 46, 3 (2003), 319–331.

[17] Bernard Chazelle. 1988. Functional Approach to Data Structures and Its Use in Multidimensional Searching. *SIAM J. Comput.* 17, 3 (1988), 427–462.

[18] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[19] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. 1994. Client-Server Paradise. In *VLDB*. 558–569.

[20] Artyom Dogtiev. 2018. *Pokémon GO Revenue and Usage Statistics (2017)*. http://www.businessofapps.com/data/pokemon-go-statistics/. Accessed: 2019-11-05.

[21] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. 1984. Scaling and Related Techniques for Geometry Problems. In *STOC*. 135–143.

[22] Volker Gaede and Oliver Günther. 1998. Multidimensional Access Methods. *Comput. Surveys* 30, 2 (1998), 170–231.

[23] Yván J. García R, Mario A. López, and Scott T. Leutenegger. 1998. A Greedy Algorithm for Bulk Loading R-trees. In *GIS*. 163–164.

[24] Michael T. Goodrich. 1999. Communication-Efficient Parallel Sorting. *SIAM J. Comput.* 29, 2 (1999), 416–432.

[25] Roberto Grossi and Giuseppe F. Italiano. 1999. Efficient Cross-trees for External Memory. In *External Memory Algorithms and Visualization*. 87–106.

[26] Ralf Hartmut Güting. 1994. An Introduction to Spatial Database Systems. *The VLDB Journal* 3, 4 (1994), 357–399.

[27] Antonin Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.

[28] Herman Haverkort and Freek V. Walderveen. 2008. Four-dimensional Hilbert Curves for R-trees. *Journal of Experimental Algorithmics* 16, Article 3.4 (2008), 19 pages.

[29] Scott Huddleston and Kurt Mehlhorn. 1982. A New Data Structure for Representing Sorted Lists. *Acta Informatica* 17, 2 (1982), 157–184.

[30] H. V. Jagadish. 1990. Spatial Search with Polyhedra. In *ICDE*. 311–319.

[31] H. V. Jagadish. 1997. Analysis of the Hilbert Curve for Representing Two-dimensional Space. *Inform. Process. Lett.* 62, 1 (1997), 17–22.

[32] Ibrahim Kamel and Christos Faloutsos. 1992. Parallel R-trees. In *SIGMOD*. 195–204.

[33] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An Improved R-tree Using Fractals. In *VLDB*. 500–509.

[34] Kothuri Venkata Ravi Kanth and Ambuj K. Singh. 1999. Optimal Dynamic Range Searching in Non-replicating Index Structures. In *ICDT*. 257–276.

[35] Nick Koudas, Christos Faloutsos, and Ibrahim Kamel. 1996. Declustering Spatial Databases on a Multi-Computer Architecture. In *EDBT*. 592–614.

[36] Scott T. Leutenegger, J. M. Edgington, and Mario A. López. 1997. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*. 497–506.
[37] Thomas Mølhave. 2012. Using TPIE for Processing Massive Data Sets in C++. *SIGSPATIAL Special* 4, 2 (2012), 24–27.
[38] Anirban Mondal, Masaru Kitsuregawa, Beng Chin Ooi, and Kian Lee Tan. 2001. R-tree-based Data Migration and Self-tuning Strategies in Shared-nothing Spatial Databases. In *GIS*. 28–33.
[39] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. 2001. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE Transactions on Knowledge and Data Engineering* 13, 1 (2001), 124–141.
[40] Nectar. 2018. *The National eResearch Collaboration Tools and Resources Project.* https://nectar.org.au/. Accessed: 2019-11-05.
[41] Yutaka Ohsawa and Masao Sakauchi. 1990. A New Tree Type Data Structure with Homogeneous Nodes Suitable for a Very Large Spatial Database. In *ICDE*. 296–303.
[42] Oracle Corporation. 2001. *Oracle Spatial User's Guide and Reference Release 9.0.1.* https://docs.oracle.com/cd/A91202_01/901_doc/appdev.901/a88805/toc.htm. Accessed: 2019-11-05.
[43] Oracle Corporation. 2019. *MySQL 8.0 Reference Manual.* https://dev.mysql.com/doc/refman/8.0/en/creating-spatial-indexes.html. Accessed: 2019-11-05.
[44] Jack A. Orenstein and T. H. Merrett. 1984. A Class of Data Structures for Associative Searching. In *PODS*. 181–190.
[45] Mark H. Overmars. 1987. *Design of Dynamic Data Structures.* Springer-Verlag.
[46] Mark H. Overmars and Jan van Leeuwen. 1981. Dynamization of Decomposable Searching Problems Yielding Good Worsts-Case Bounds. In *5th GI-Conference on Theoretical Computer Science*. 224–233.
[47] Mark H. Overmars and Jan van Leeuwen. 1981. Worst-case Optimal Insertion and Deletion Methods for Decomposable Searching Problems. *Inform. Process. Lett.* 12, 4 (1981), 168–173.
[48] Apostolos Papadopoulos and Yannis Manolopoulos. 2003. Parallel Bulk-loading of Spatial Data. *Parallel Comput.* 29, 10 (2003), 1419–1444.
[49] Ben Popper. 2017. *Google Announces over 2 Billion Monthly Active Devices on Android.* https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users. Accessed: 2019-11-05.
[50] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2018. Theoretically Optimal and Empirically Efficient R-trees with Strong Parallelizability. *Proceedings of the VLDB Endowment* 11, 5 (2018), 621–634.
[51] Jianzhong Qi, Rui Zhang, Lars Kulik, Dan Lin, and Yuan Xue. 2012. The Min-dist Location Selection Query. In *ICDE*. 366–377.
[52] Nick Roussopoulos and Daniel Leifker. 1985. Direct Spatial Search on Pictorial Databases Using Packed R-trees. In *SIGMOD*. 17–31.
[53] James B. Saxe and Jon L. Bentley. 1979. Transforming Static Data Structures to Dynamic Structures. In *FOCS*. 148–168.
[54] Bernd Schnitzer and Scott T. Leutenegger. 1999. Master-client R-trees: A New Parallel R-tree Architecture. In *SSDBM*. 68–77.
[55] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB*. 507–518.
[56] Apache Spark. 2016. *Spark Programming Guide.* https://spark.apache.org/docs/1.6.0/programming-guide.html. Accessed: 2019-11-28.
[57] Yufei Tao, Wenqing Lin, and Xiaokui Xiao. 2013. Minimal MapReduce Algorithms. In *SIGMOD*. 529–540.
[58] The Pokémon Company. 2018. *Pokémon Go.* http://www.pokemongo.com. Accessed: 2019-11-05.
[59] United States Census Bureau. 2006. *TIGER/Line Shapefiles and TIGER/Line Files.* https://www.census.gov/geo/maps-data/data/tiger-line.html. Accessed: 2019-11-05.
[60] Pan Xu and Srikanta Tirthapura. 2014. Optimality of Clustering Properties of Space-Filling Curves. *ACM Transactions on Database Systems* 39, 2 (2014), 10:1–10:27.
[61] Simin You, Jianting Zhang, and Le Gruenwald. 2013. Parallel Spatial Query Processing on GPUs Using R-trees. In *2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. 23–31.
[62] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Conference on Hot Topics in Cloud Computing*. 10–10.
[63] Rui Zhang, Panos Kalnis, Beng Chin Ooi, and Kian-Lee Tan. 2005. Generalized Multidimensional Data Mapping and Query Processing. *ACM Transactions on Database Systems* 30, 3 (2005), 661–697.
[64] Rui Zhang, Beng Chin Ooi, and Kian-Lee Tan. 2004. Making the Pyramid Technique Robust to Query Types and Workloads. In *ICDE*. 313–324.
[65] Rui Zhang, Jianzhong Qi, Martin Stradling, and Jin Huang. 2014. Towards a Painless Index for Spatial Objects. *ACM Transactions on Database Systems* 39, 3 (2014), 19:1–19:42.

Received February 2019; revised January 2020; accepted April 2020