Theoretically Optimal and Empirically Efficient R-trees with Strong Parallelizability

Jianzhong Qi[1], Yufei Tao[2], Yanchuan Chang[1], Rui Zhang[1]

[1]School of Computing and Information Systems, The University of Melbourne
jianzhong.qi@unimelb.edu.au, yanchuanc@student.unimelb.edu.au, rui.zhang@unimelb.edu.au

[2]Department of Computer Science and Engineering, Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

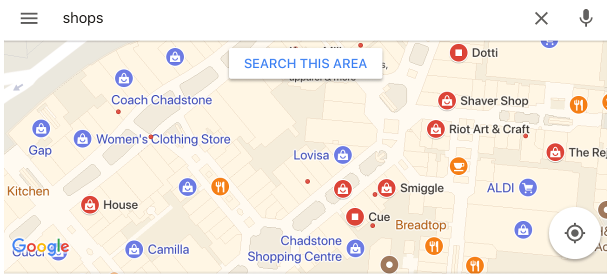
ABSTRACT

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 bulk-loading, and over both practical and worst-case workloads. To address this need, we revisit a classic multidimensional access method – the R-tree. We propose a novel R-tree packing strategy that produces R-trees with an asymptotically optimal I/O complexity for window queries in the worst case. Our experiments show that the R-trees produced by the proposed strategy are highly efficient on 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. Experimental results confirm the efficiency and scalability of the parallel algorithm over large data sets.

PVLDB Reference Format:
Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. Theoretically Optimal and Empirically Efficient R-trees with Strong Parallelizability. *PVLDB*, 11(5): xxxx-yyyy, 2018.
DOI: https://doi.org/10.1145/3177732.3177738

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 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 [1], 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., "pokémons") 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.



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

(b) Gaming objects in a game view

Figure 1: Window queries in real applications

In these applications, there may be hundreds of millions of spatial objects (e.g., shops, restaurants, pokémons, etc.) with a variety of distributions to be managed. Meanwhile, there may be millions of service requests from users, e.g., Google Maps is answering millions of queries per day [2], and Pokémon GO is attracting over 30 million active users [3]. Reporting POIs or pokémons 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* [29]. Our aim is 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 [12, 15, 17, 30, 34, 45] and industrial applications [4, 5]. An R-tree is a balanced

Proceedings of the VLDB Endowment, Vol. 11, No. 5
Copyright 2018 VLDB Endowment 2150-8097/18/01... $ 10.00.
DOI: https://doi.org/10.1145/3177732.3177738

tree structure designed 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 *capacity* of the node, i.e., the maximum number of entries per node, denoted as B. Given an R-tree, a window query returns all the data objects (e.g., POIs or pokémons) 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 [17, 34, 37, 45]. However, all these heuristics suffer from the limitation that they do not produce an R-tree with attractive performance guarantees in the worst case. The *Priority R-tree* (PR-tree) [15] is an R-tree structure that has a 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 [12]. 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 [30], 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; it also suffers in bulk-loading performance; 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, as a recent study [52] shows. We target application scenarios such as digital mapping, where the data sets are relatively static and can be batch-updated. We construct R-trees that are query cost optimal for static data. Our R-trees handle dynamic data updates efficiently as well, although the optimality may be compromised. We also offer efficient techniques for periodical R-tree rebuilding to preserve 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 the R-trees produced 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. As a result, we obtain data points with no repetitive coordinates in either dimension. We then simply pack every B data points into a leaf node (except possibly the last leaf node) of an R-tree in the 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* [41] which is a common type of *space-filling curves* (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 the ascending order of the Z-cure 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.

Since our R-tree packing strategy relies only on sorting, it takes $O((n/B)\log_{M/B}(n/B))$ 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 [13], which paves the foundation for designing algorithms for MapReduce systems [22]. 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 $O(n \log n)$. For modern machines, s is large, e.g., at the order of millions, allowing the proposed algorithm to bulk-load an R-tree with a very large number of data points in just a few rounds of computation.

While the rank space has been used by the computational geometry community to develop theoretical bounds [21, 24], 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 indexes were previously thought to have poor worst-case query costs. Indeed, as shown in [15], 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 [32, 39, 47] 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 that relies only on sorting. We propose such an algorithm under the massively parallel communication model (and thus, works on MapReduce systems) with attractive performance guarantees.

3. 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 44% lower than that of PR-trees [15] and similar to that of STR-trees [37], 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 12% lower than that of PR-trees and 20% lower than that of STR-trees. The proposed bulk-loading algorithm also outperforms the PR-tree bulk-loading algorithm in running time by 86% over large data sets with 20 million data points.

The rest of the paper is organized as follows: Section 2 discusses 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 studies the empirical performance of the proposed algorithms. 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 queries [51]. 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 [29]. For a comprehensive review on spatial indices, interested readers are referred to [25].

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 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 2 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 [17, 19, 31, 40, 45] propose heuristics to optimize the node MBRs during dynamic data insertion. The *R*-tree* [17], 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 [23, 30, 34, 37, 43] 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 [43] sort the data objects by their x-coordinates and pack every B objects into a leaf node. Leutenegger et al. [37] 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* [23, 30, 34]. R-trees bulk-loaded based on Hilbert ordering have good window query performance on nicely distributed data [30]. However, these R-trees do not have worst-case optimal query performance.

There are also top-down bulk-loading algorithms, e.g., *Top-down Greedy Split* (TGS) [26]. 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 $O((n/B)\log_2 n)$ [15].

Agarwal et al. [12] propose an algorithm to bulk-load a *Box-tree*, which can be converted to an R-tree that obtains 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* [15] 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 [12]. 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. [15] further propose a bulk-loading strategy which lowers the I/O cost to $O((n/B)\log_{M/B}(n/B))$. The main issues of the PR-tree are that it lacks practical efficiency in bulk-loading and answering queries with small query windows [30].

We also note that other spatial indices such as *kd-trees* [18], *O-trees* [35], and *cross-trees* [28] 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. [13] propose a randomized algorithm that can bulk-load a kd-tree with $O(\text{poly}\log_s n)$ 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 [11]. These indices are not discussed further.

Parallel R-tree management. Parallelism has been exploited to scale R-trees to large data sets and user groups. An early study [33] 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 [36, 38, 44] assume a shared nothing (client-server) architecture for distributed R-tree storing and query processing. Koudas et al. [36] store the inner nodes on a server while the leaf nodes on clients. Schnitzer and Leutenegger [44] further create local R-trees

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 q
k	The answer set size of a window query
B	The node capacity of an R-tree
h	The height of an R-tree
g	The number of machines in a cluster
s	The number of data points allowed in a machine

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 [42] 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 which serves as a coordinator for query processing. A more recent study [10] bulk-loads an R-tree with the MapReduce framework level by level, where each level takes one MapReduce round. The study also uses an SFC for object ordering. However, its focus is on deciding the split points among tree nodes. The tree nodes can contain varying numbers of entries, which leads to non-optimal bulk-loading costs. Similar ideas have been used on GPUs [48] without a cost analysis.

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 the following discussion, although our approach also applies to any $d > 2$. We focus on window query processing. 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

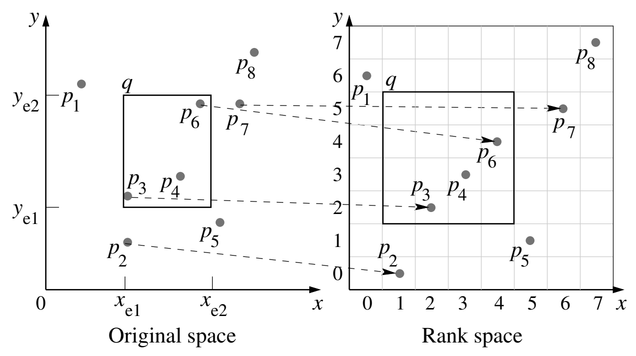


Figure 2: Mapping to rank space

Before creating an index structure over P, we first map the data points into a 2-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. Rank ties in a dimension are broken by the coordinates in the other dimension of the original space. We assume no data points with the same coordinates in both dimensions.

Define by $[n]$ the integer domain $[0, n-1]$. After the mapping, P *becomes a set of* n 2*-dimensional points in* $[n]^2$ *such that no two points share the same* x*- or* y*-coordinate.*

Figure 2 illustrates the mapping with a set of 8 ($n = 8$) points $P = \{p_1, p_2, ..., p_8\}$. 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, p_2 has an x-coordinate of 1 while p_3 has an x-coordinate of 2, because p_2 has a smaller y-coordinate in the original space.

A query rectangle $q = [x_{e1}, x_{e2}] \times [y_{e1}, y_{e2}]$ in the original space is mapped to a rectangle $q = [x_1, x_2] \times [y_1, y_2]$ in $[n]^2$. Here, x_1 (y_1) is the smallest rank of the data points whose x-coordinates (y-coordinates) in the original space are greater than or equal to x_{e1} (y_{e1}); x_2 (y_2) is the largest rank of the data points whose x-coordinates (y-coordinates) in the original space are smaller than or equal to x_{e2} (y_{e2}). In Fig. 2, the solid rectangle represents a query rectangle q. In the original space, the query range $[x_{e1}, x_{e2}]$ 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, $[x_{e1}, x_{e2}]$ is mapped to $[1, 4]$ in the rank space. Similarly, the query range $[y_{e1}, y_{e2}]$ 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.

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* [41] 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 $[n]^2$. Suppose that $p = (x, y)$, where $x = \alpha_1\alpha_2...\alpha_l$ and $y = \beta_1\beta_2...\beta_l$ in binary form where $l = \log_2 n$. Then, $Z(p) = \beta_1\alpha_1\beta_2\alpha_2...\beta_l\alpha_l$. 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, where $B \geq 1$ is a parameter that controls how many points 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.

Figure 3 illustrates the R-tree packing strategy. 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

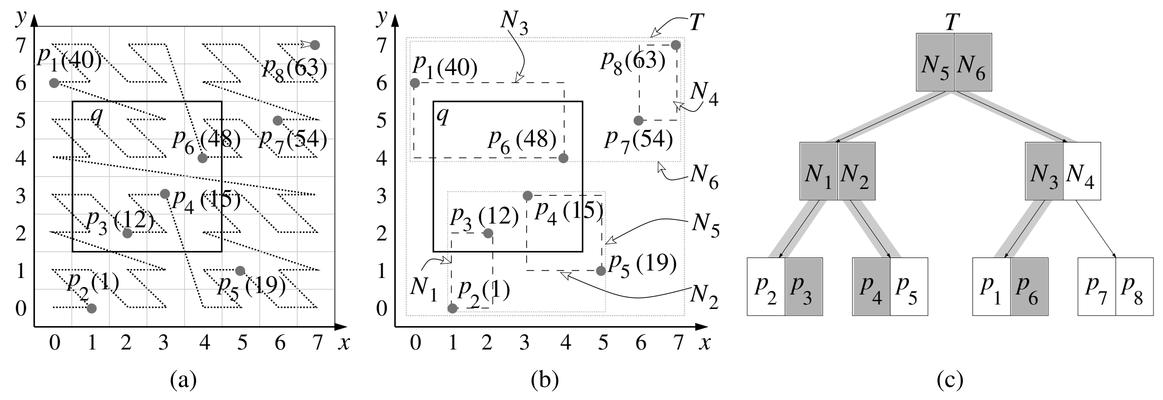


Figure 3: R-tree packing

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 $N_5 = \langle N_1, N_2\rangle$ and $N_6 = \langle N_3, N_4\rangle$. A root node is further created to point to N_5 and N_6. Figures 3b and 3c show the MBRs and the R-tree T created.

Algorithm 1: Build-R-tree

```
Input: $P = \{p_1, p_2, ..., p_n\}$: 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 $T$;
```

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 (i) two sorts on the data points to map them into the rank space (Line 1), (ii) a linear scan on the data points to compute their Z-order values (Lines 2 and 3), (iii) another sort on the Z-order values (Line 4), and (iv) 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.

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 two B-trees to store pairs of point coordinates in the original space and corresponding coordinate in the rank space, one for each dimension. 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 [12, 20]. Let $h \le \log_B n$ be the height of the tree. Label the levels of the tree as $1, 2, ..., h$ 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 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. 4a, 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.

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 as 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. 4a to illustrate the proof for the case where $t = 1$, i.e., at the leaf node level. It shows the four leaf

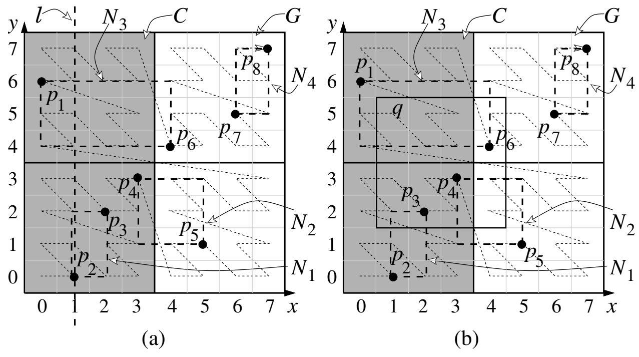


Figure 4: Window query I/O cost

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, which is represented by the black solid line grid in the figure. 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. 4a, 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, N_3 intersects the line, and $X(N_3) = [0, 4]$. Such 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 $2n/m \le 2\sqrt{n/B^t}$ 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. 4a, 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. 4a, $b-a+1 = 3-0+1 = 4$. 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 $h \le \log_B n$, line ℓ interests the MBRs of $O(\sqrt{n/B}) + O(\sqrt{n/B^2}) + ... + O(\sqrt{n/B^h}) = O(\sqrt{n/B})$ 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 $2^\rho \le 2n$. The above argument can also be generalized to an arbitrary fixed dimensionality $d \ge 2$ 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 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 $q = [x_1, y_1] \times [x_2, y_2] \times ... \times [x_d, y_d]$ in $[n]^d$, where $d \ge 2$ is an arbitrary fixed dimensionality. For each $i \in [1, d]$, set $\lambda_i = y_i - x_i + 1$, and define $Z_i = \{1, 2, ..., d\} \setminus \{i\}$, namely, Z_i includes all the integers from 1 to d except i. We will prove a stronger version of our previous lemma: our structure answers the query in

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

I/Os where $\Lambda = \sum_{i=1}^{d} \prod_{j \in Z_i} \lambda_j$. In this bound, $\log_B n$, $\Lambda^{1/d}/B^{1-1/d}$, 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 inside 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 $O(\log_B n + \sqrt{(\lambda_1 + \lambda_2)/B} + k/B)$, while for $d = 3$, the cost becomes $O(\log_B n + (\lambda_1\lambda_2 + \lambda_1\lambda_3 + \lambda_2\lambda_3)^{1/3}/B^{2/3} + k/B)$. Since $\lambda_i \le n$ for all $i \in [1, n]$, it always holds that $\prod_{j \in Z_i} \lambda_j \le n^{d-1}$ and $\Lambda \le d \cdot n^d$. 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 will prove the following lemma, which is sufficient for establishing our claim.

LEMMA 2. *The query rectangle q partially intersects the MBRs of $O(1 + \Lambda^{1/d}/(B^t)^{1-1/d})$ nodes at level t.*

Proof. Let m be the smallest power of 2 at least $(\Lambda \cdot B^t)^{1/d}$. Divide $[n]^d$ into an $\underbrace{(n/m) \times (n/m) \times ... \times (n/m)}_{d}$ grid G, where each cell 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 $j \neq i$.

We use Fig. 4b to illustrate the proof, where $d = 2$ and $t = 1$, i.e., at the leaf node level. We have $q = [1, 4] \times [2, 5]$ (the solid line rectangle) and hence $\lambda_1 = 4 - 1 + 1 = 4$ and $\lambda_2 = 5-2+1 = 4$. Thus, $m = (\Lambda \cdot B^t)^{1/d} = \sqrt{(\lambda_1 + \lambda_2)B^t} = \sqrt{(4+4) \times 2^1} = 4$. 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 (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. 4b). We will prove that there can be at most $O(1 + m/B^t + \prod_{j \in Z_i} \lambda_j/m^{d-1})$ nodes intersecting each of the two faces of q perpendicular to dimension i. Summing this up on all d dimensions gives an upper bound on the total number of nodes that partially intersect q:

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

as desired in the lemma.

Due to symmetry, it suffices to consider the face of q that corresponds to x_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 $[a, b] \times \underbrace{[n] \times [n] \times ... \times [n]}_{d-1}$ for some a, b satisfying $b - a + 1 = m$ and b is a multiple of 2. In Fig. 4b, 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., N_3 intersects the left edge of q, and $X(N_3) = [0, 4]$. Such u can be 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 $O(\prod_{j=2}^{d} \lceil \lambda_i/m \rceil) = O(1+\lambda_2\lambda_3...\lambda_d\ /m^{d-1})$ times within the dimension-1 left face of q. This is because there are $O(\prod_{j=2}^{d} \lceil \lambda_i/m \rceil)$ cells of C within the range of q in dimensions 2 to d (e.g., in Fig. 4b, there are $1+\lambda_2/m = 1+4/4 = 2$ 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 1, there are at most $O(\prod_{j=2}^{d} \lceil \lambda_i/m \rceil)$ 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 $O(1 + \lambda_2\lambda_3...\lambda_d/m^{d-1})$.
- 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 $O(1 + m/B^t + \prod_{j \in Z_i} \lambda_j/m^{d-1})$ nodes at level t. This completes the proof. $\square$

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. 4a, 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. 4a, 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. 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 [13, 14, 16]. Popular parallel frameworks such as MapReduce [22] and Spark [49] are typical examples of this model.

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. We consider only algorithms that require a machine to receive/send $O(s)$ words of information in each round (with the terminology of [16], these algorithms must have *load* $O(s)$).

MPC algorithms are measured by: (i) the *number of computation rounds* $\mathcal{R}$, (ii) the (parallel) *running time* $\mathcal{T}$, and (iii) the *total amount of computation* $\mathcal{W}$. The running time sums up the maximum computation cost of a *single machine* in each round; the total amount of computation 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,

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

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, and $O(n \log n)$ total amount of computation [27] (see Tao et al. [46] for a simple algorithm when $s \geq g \ln(g \cdot n)$ holds).

Mapping n data points to the rank space and sorting by 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.

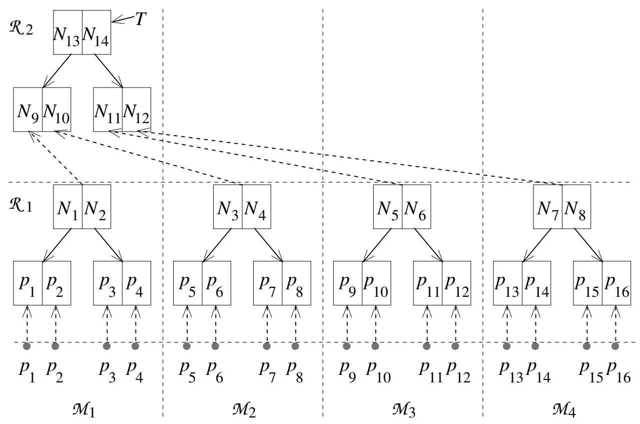

$\mathcal{R}_2$
N_{13} N_{14}
T
N_9 N_{10} N_{11} N_{12}
$\mathcal{R}_1$
N_1 N_2 N_3 N_4 N_5 N_6 N_7 N_8
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14} p_{15} p_{16}
$\mathcal{M}_1$ $\mathcal{M}_2$ $\mathcal{M}_3$ $\mathcal{M}_4$

Figure 5: Parallel R-tree bulk-loading

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.

Figure 5 illustrates the rounds, 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

Our R-tree structure can also allow dynamic data updates. An object to be removed can be simply deleted from our R-tree in the same way as for a conventional R-tree. An object to be inserted need to be first mapped to the rank space in a similar way to that of query window mapping. Let p be an object to be inserted and (x_e, y_e) be its coordinates in the original space. If x_e (y_e) equals to the x-coordinates (y-coordinates) of some existing data points, x_e (y_e) is simply mapped to the largest x-rank (y-rank) of those data points. Otherwise, x_e (y_e) is mapped to two coordinates x_1 and x_2 (y_1 and y_2) in the rank space. Here, x_1 (y_1) is the smallest rank of the data points whose x-coordinates (y-coordinates) in the original space are greater than x_e (y_e); x_2 (y_2) is the largest rank of the data points whose x-coordinates (y-coordinates) in the original space are smaller than x_e (y_e). The four coordinates x_1, x_2, y_1, y_2 may form four points (x_1, y_1), (x_1, y_2), (x_2, y_1), and (x_2, y_2) in the rank space. To ensure no false negatives in query processing, object p needs to be mapped to all these points. Then, it can be inserted into our R-tree in the same way as for a conventional R-tree. Allowing dynamic data updates compromises the optimal worst-case I/O cost. A periodic rebuild of the R-tree is needed for optimal performance.

We target applications where the data set is relatively static, e.g., buildings in a map tend not to change in every second. The parallel bulk-loading algorithm proposed in Section 4 can be used to periodically rebuild the R-tree to cope with data updates. As our experiments show, it only takes 25.7 minutes to build an R-tree with 100 million data objects. This should satisfy the targeted applications.

6. EXPERIMENTS

This section reports experimental results on comparing the bulk-loading and window query performance of the R-tree constructed by the proposed strategy with those of the STR-tree [37], Hilbert R-tree [30], TGS R-tree [26], and PR-tree [15], which have been described in Section 2. We denote the baseline algorithms by "**STR**", "**HR**", "**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

Table 2: Parameters and Their Settings

Parameter	Setting
Distribution	**Uniform**, Skew, Cluster
n	0.5M, 1M, 5M, **10M**, 20M
α	**1**, 3, 5, 7, 9
Query window area (%)	0.005, **0.01**, 0.05, 0.1, 0.5, 1, 2

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.

Following previous studies [15, 26, 30, 37], we focus on the I/O cost of the algorithms.

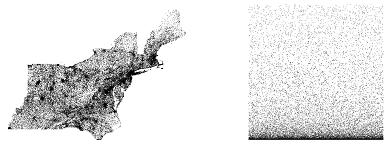
6.1 Experimental Setup

The window query experiments are run on a 64-bit machine running Ubuntu 14.04 with a 2.60 GHz Intel i5 CPU, 4 GB memory, and a 100 GB hard disk. We use Ke Yi's single-machine implementation [6] of the Hilbert R-tree, TGS R-tree, and PR-tree, which uses the TPIE library [7] – 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 STR-tree and the proposed HRR and ZR R-trees using TPIE. In all the R-tree structures, 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. We use a block size of 4 KB, and the maximum fanout of a node is 102.

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 16 virtual-node cluster created from an academic computing cloud (Nectar [8]) running on OpenStack. Each virtual-node has 12 GB memory and 4 cores running at 2.6 GHz. One of the nodes acts as the master and the other 15 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.

We use both real and synthetic data sets. The real data set contains 17,468,292 rectangles (950 MB in size) representing geographical features in 18 eastern states of the USA extracted from the TIGER/Line 2006SE data [9]. We use the center of the rectangles as our data points. We denote this data set as "**Tiger**". Figure 6a illustrates the data set.

Synthetic data sets are generated with a space domain of 1×1 where the data set cardinality ranges from 0.5 to 20 million. We generate three groups of synthetic data sets, denoted as "**Uniform**", "**Skew**", and "**Cluster**", respectively. The Uniform data sets contain data points with uniform distribution. The Skew and Cluster data sets are



(c) Cluster data

Figure 6: Experimental data

generated following the PR-tree paper [15]. 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}) where $\alpha \geq 1$. Figure 6b illustrates a Skew data set where $\alpha = 9$. The Cluster data set contains 10,000 clusters with centers evenly distributed on a horizontal line. Each cluster has 1,000 points following a uniform distribution in a 0.00001×0.00001 square around the center. Figure 6c illustrates a portion of the Cluster data set with four clusters.

We vary query window size, data set size, and data skewness in the experiments. The experimental parameters are summarized in Table 2, where default values are in bold.

6.2 Results

We present the experimental results in this subsection.

6.2.1 Window Query Processing

We start with the window query performance of the R-trees. We generate 100 queries at random locations in each experiment. For ease of comparison, we follow previous studies [15, 26, 30, 37] 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.

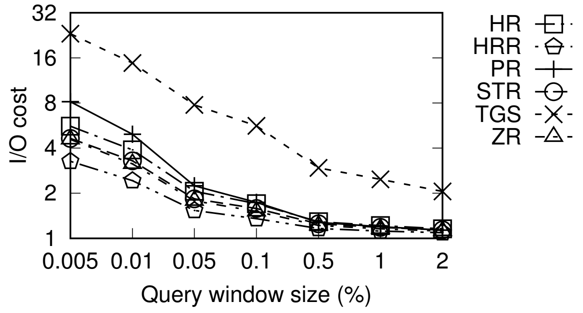


Figure 7: Query I/O cost on Uniform data (varying query window size)

Varying query window size. We first vary the area of the query window from 0.005% to 2% of the data space. Figure 7 shows the query I/O cost relative to the output size

k/B over 10 million uniform data points. A general observation is that the relative query costs of the R-trees created by the various packing strategies decrease as the query window area increases. This is because a larger query window overlapping a tree node has a better chance of overlapping the data points in this node, i.e., there are lower percentages of extra query I/Os that do not contribute to the output.

Meanwhile, the R-trees HRR and ZR created by the proposed packing strategy have the smallest query I/O costs. The advantage is more significant with smaller query windows, e.g., when the query window area is 0.005% of the data space, the query I/O costs of HRR and ZR are 60% and 44% lower than that of PR, respectively (note the logarithmic scale). HRR and ZR also outperform HR, STR, and TGS. This demonstrates that HRR and ZR not only have an asymptotically optimal cost in the worst case but also perform well in other cases. HRR in particular outperforms HR by up to 42% when the query window area is 0.005%. This advantage attributes to the rank space mapping before packing the data points. The proposed packing strategy effectively incorporates the designs of both HR and STR, which have reported good performance 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 [34] that compares the query performance of R-trees packed with the Hilbert curve and the Z-curve in the same Euclidean space.

To help further understand the benefit of the proposed packing strategy, we list the average output size (k/B) per query as follows. For the seven different query window areas tested (i.e., from 0.005% to 2% of the data space size), the output sizes are 4.96, 9.81, 48.44, 96.49, 466.82, 925.08, and 1815.40, respectively. Based on these output sizes and the relative query I/O costs shown in Fig. 7, 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, PR, and ZR are 1.09, 1.13, and 1.11, which correspond to 1978.79 (1815.40×1.09), 2051.40 (1815.40×1.13), and 2015.09 (1815.40×1.11) I/Os, respectively. This means that HRR and ZR have 72.61 and 36.31 fewer I/Os than PR, respectively. While these numbers might seem small, they 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.

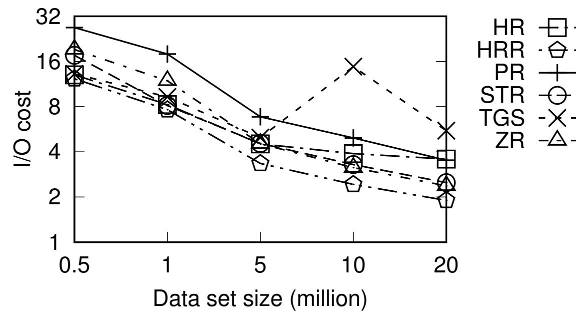


Figure 8: Query I/O cost on Uniform data (varying data set cardinality)

Varying data set cardinality. Next, we vary the data set cardinality from 0.5 to 20 million while keeping the query window size at 0.01% of the data space (output size ranging from 0.49 to 19.54). We see from Fig. 8 that HRR and ZR outperform PR consistently by reducing about 50% and 30% of the relative query I/O costs, respectively. For fairness, the PR-tree is designed for rectangles. It may not be optimal on point data for which HRR and ZR are designed. HR and STR have closer query performance to that of HRR and ZR, for that they share similar design with HRR and ZR. However, ZR still has a lower cost when the data set cardinality exceeds one million, while HRR has the lowest cost constantly. This again demonstrates the advantage of using the rank space for indexing. The performance of TGS fluctuates the most. It relies on a heuristic optimization function when packing the data points, i.e., minimizing the area of the MBRs, which may not be optimal for all cases.

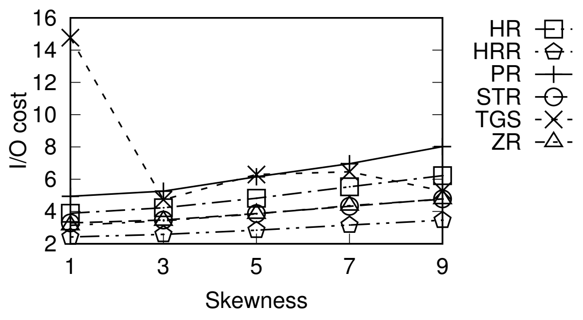


Figure 9: Query I/O cost on Skew data

Varying skewness. Figure 9 shows the query I/O performance on Skew data where the skewness parameter α is varied from 1 to 9 (output size ranging from 9.81 to 4.75). As the skewness increases, the relative I/O costs of all the R-trees increase except for the TGS R-tree. This is natural as more skewed data makes it more difficult to reduce overlaps between the MBRs of different tree nodes, leading to more blocks being accessed for query processing. TGS is an exception. Its MBR minimizing optimization function leads to better packing for more skewed data. Regardless of this, HRR and ZR still outperform TGS and the other packing strategies. Their relative query I/O costs are up to 57% and 40% lower than those of PR, respectively. STR is again the closest for that it shares a similar packing strategy with ZR.

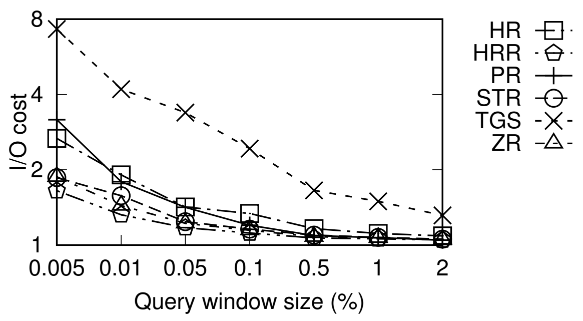


Figure 10: Query I/O cost on Tiger data

Tiger data. Figure 10 shows the relative query I/O cost on Tiger data where the query window size is varied (output size ranging from 7.77 to 3950.92). The query performance comparison of the different R-tree packing strategies is very similar to that on Uniform data (Fig. 7). HRR and ZR again have the smallest relative query I/O costs. HRR outperforms PR consistently, while ZR outperforms PR when the query window area is within 1% of the data space, and the advantage is up to 41% when the query window area is 0.005% of the data space. HRR and ZR also outperform HR, STR, and TGS. This confirms the superiority of HRR and ZR in that they not only have an optimal cost in the worst case but also perform well in practice.

Cluster data. The Cluster data set is designed to test the worst-case performance of the R-trees. Following the PR-tree paper [15], we generate long and thin window queries each with an area of 10^{-7} 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 area 10^{-7} divided by the query width. Such queries have an average output size of 980.23.

Table 3: Query I/O Cost on Cluster Data

Tree	HR	HRR	PR	STR	TGS	ZR
I/O cost	102.57	**1.46**	2.00	2.21	2.32	1.76

Table 3 shows the query I/O cost relative to the output size k/B. Since the Cluster data set was designed to reflect the worst-case scenario, the query performance of all packing strategies is worse than that on the previous data sets tested. The query performance of HR degrades the most, as HR does not have a bound on worst-case performance. In contrast, HRR still has the lowest query I/O cost, which attributes to the rank space mapping. HRR, PR, and ZR are all asymptotically optimal in the worst case. They show the best performance on this data set, with HRR and ZR outperforming PR by 27% and 12%, respectively. STR is not worst-case optimal, but its packing strategy resembles that of ZR, and hence its performance is only 20% worse than that of ZR. TGS benefits from its top-down optimization strategy, but still has a 32% higher cost than ZR.

6.2.2 *Bulk-loading*

This subsection reports the R-tree bulk-loading performance. We implemented both the standalone bulk-loading algorithm described in Section 3.2 and the parallel bulk-loading algorithm described in Section 4.2. For the standalone algorithm, we measure both the I/O and the response time (denoted as "**ZR**"). For the parallel algorithm, we measure (i) the response time (denoted as "**ZR-R**"), which is the duration for which the algorithm runs, and (ii) the running time $\mathcal{T}$ (denoted as "**ZR-M**"), which is the sum of the maximum single machine response time over all MapReduce rounds, and (iii) the communication time (denoted as "**ZR-C**"), which is the part of the response time spent on communication. For comparison, we also implemented a level-by-level parallel bulk-loading algorithm based on the proposed packing strategy and measured its response time (denoted as "**L-R**"), running time (denoted as "**L-M**"), and communication time (denoted as "**L-C**"). 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 TPIE.

We also implemented the bulk-loading algorithm for the proposed packing strategy using the Hilbert curve. We denote the standalone implementation as "**HRR**". As Figs. 11 and Fig. 12a 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.

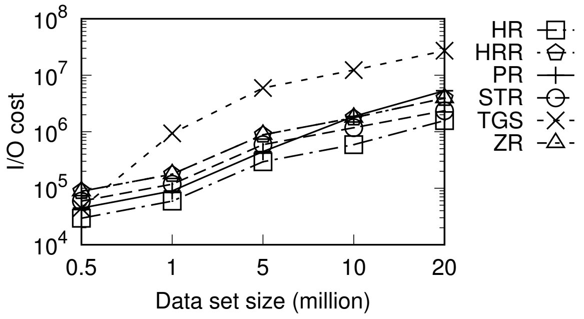


Figure 11: Bulk-loading I/O cost on Uniform data

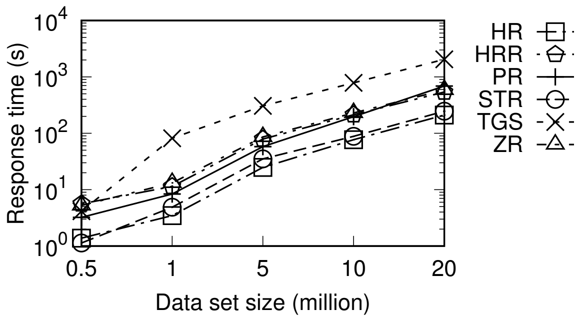
Varying data set cardinality. We first vary the data set cardinality. Figure 11 shows the bulk-loading I/O costs, which increase with the data set cardinality as expected. The advantage of HRR and ZR in bulk-loading I/O cost is less obvious compared with that in query I/O cost. Both HR and STR outperform 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. Even though HR and STR have lower bulk-loading costs, their query performance can be much worse than that of HRR and ZR, 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. The I/O costs of HRR and ZR become smaller 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 the highest bulk-loading I/O cost. This is due to its repetitive data access for optimization function computation.

Figure 12a shows the bulk-loading time of the standalone algorithms. The comparative performance of the algorithms is similar to that on I/O costs (Fig. 11). The running time of HRR and ZR grows slower than that of PR; HRR and ZR outperform PR for data set cardinality over 10 million.

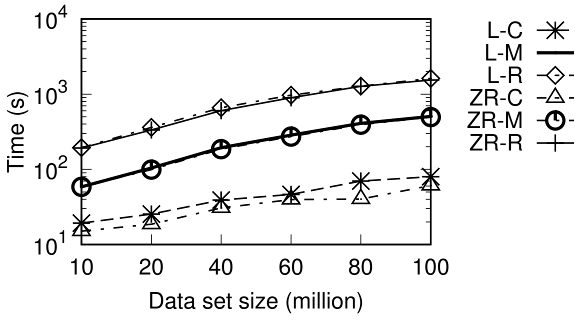
Figure 12b shows the communication time (ZR-C), running time (ZR-M), and response time (ZR-R) of the proposed parallel bulk-loading algorithm. We scale the parallel algorithm to 100 million points. ZR-C, ZR-M, and ZR-R are consistently smaller than their level-by-level counterparts L-C, L-M, and L-R. ZR-C is up to 42% smaller than L-C due to the smaller number of communication rounds of the proposed algorithm, while ZR-M is only up to 5% smaller than L-M since both algorithms perform similar computations. Together, the overall response time ZR-R is up to 9% smaller than L-R. 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

Table 4: Bulk-loading Time Cost on Tiger and Cluster Data (Second)

	HR	HRR	PR	STR	TGS	ZR	ZR-C	ZR-M	ZR-R	L-C	L-M	L-R
Tiger	202.84	588.99	530.06	192.47	1934.73	516.31	28.41	**133.69**	487.17	31.23	145.34	501.76
Cluster	**76.21**	256.78	196.04	86.58	895.90	222.35	15.19	80.12	276.54	23.32	84.67	290.27



(a) Standalone algorithms



(b) Distributed algorithms

Figure 12: Bulk-loading time cost on Uniform data

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. 12a and 12b, we see that, on 10 million data points, both the response time (ZR-R, 192.56 seconds) and the running time (ZR-M, 58.44 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 70%, and this advantage grows with the data set size (e.g., 86% on 20 million data points), demonstrating the scalability of the proposed parallel algorithm.

Tiger and Cluster data. Table 4 shows the algorithm running times on Tiger and Cluster data. The comparative performance of the algorithms is again similar to that on Uniform data. The advantage of the proposed bulk-loading strategy becomes more significant as there are more data points. On Tiger data (17 million points), ZR-M is the smallest, while ZR and ZR-R are smaller than PR. On Cluster data (10 million points), ZR-M is close to both HR and STR, while ZR is close to PR. This demonstrates the robustness of the proposed bulk-loading strategy. The level-by-level parallel bulk-loading algorithm has both higher response time (L-R) and running time (L-M) than those of the proposed parallel bulk-loading algorithm. This again justifies the advantage of the proposed algorithm over the simple level-by-level parallel bulk-loading algorithm.

Experiments on Skew data show similar patterns, and the results are omitted for conciseness.

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 practically 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 [15, 52]. 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 44% 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 12% lower than that of PR-trees and 20% 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 86% on large data sets with 20 million data points.

For future work, we plan to investigate algorithms to handle data updates without compromising the worst-case optimal query performance. Also, applying the rank space mapping technique over other spatial indices such as quad-trees and GiMP [50] to obtain worst-case optimal query performance would be an interesting direction to explore.

8. ACKNOWLEDGMENTS

This work is supported by Australian Research Council (ARC) Future Fellowships Project FT120100832 and Discovery Project DP130104587, The University of Melbourne Early Career Researcher Grant (Project Number: 603049), a direct grant (Project Number: 4055079) from The Chinese University of Hong Kong, and a Faculty Research Award from Google.

9. REFERENCES

[1] http://www.pokemongo.com.

[2] http://en.wikipedia.org/wiki/Google_Search.

[3] https://www.wired.com/2016/09/pokemon-go-just-fine-without/.

[4] http://www.microsoft.com/sqlserver/2008/en/us/spatial-data.aspx.

[5] http://download.oracle.com/otndocs/products/spatial/pdf/spatial_features_jsirev.pdf.

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

[7] http://madalgo.au.dk/tpie/.

[8] https://nectar.org.au/.

[9] https://www.census.gov/geo/maps-data/data/tiger-line.html.

[10] D. Achakeev, M. Seidemann, M. Schmidt, and B. Seeger. Sort-based parallel loading of r-trees. In *1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 62–70, 2012.

[11] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *28th International Colloquium on Automata, Languages and Programming*, pages 115–127, 2001.

[12] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. J. Haverkort. Box-trees and r-trees with near-optimal query time. In *17th Annual Symposium on Computational Geometry (SCG)*, pages 124–133, 2001.

[13] P. K. Agarwal, K. Fox, K. Munagala, and A. Nath. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *PODS*, pages 429–440, 2016.

[14] A. Andoni, A. Nikolov, K. Onak, and G. Yaroslavtsev. Parallel algorithms for geometric graph problems. In *STOC*, pages 574–583, 2014.

[15] L. Arge, M. D. Berg, H. Haverkort, and K. Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Transactions on Algorithms*, 4(1):9:1–9:30, 2008.

[16] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *PODS*, pages 273–284, 2013.

[17] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[18] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[19] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The x-tree : An index structure for high-dimensional data. In *VLDB*, pages 28–39, 1996.

[20] M. Berg, M. Kreveld, M. Overmars, and O. C. Schwarzkopf. *Computational Geometry*. Springer Berlin Heidelberg, 2000.

[21] B. Chazelle. Functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[22] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-server paradise. In *VLDB*, pages 558–569, 1994.

[24] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *STOC*, pages 135–143, 1984.

[25] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[26] Y. J. García R, M. A. López, and S. T. Leutenegger. A greedy algorithm for bulk loading r-trees. In *6th ACM International Symposium on Advances in Geographic Information Systems*, pages 163–164, 1998.

[27] M. T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 1999.

[28] R. Grossi and G. F. Italiano. Efficient cross-trees for external memory. In J. M. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 87–106, 1999.

[29] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[30] H. Haverkort and F. V. Walderveen. Four-dimensional hilbert curves for r-trees. *Journal of Experimental Algorithmics*, 16:1–19, 2008.

[31] H. V. Jagadish. Spatial search with polyhedra. In *ICDE*, pages 311–319, 1990.

[32] H. V. Jagadish. Analysis of the hilbert curve for representing two-dimensional space. *Information Processing Letters*, 62(1):17–22, 1997.

[33] I. Kamel and C. Faloutsos. Parallel r-trees. In *SIGMOD*, pages 195–204, 1992.

[34] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB*, pages 500–509, 1994.

[35] K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *ICDT*, pages 257–276, 1999.

[36] N. Koudas, C. Faloutsos, and I. Kamel. Declustering spatial databases on a multi-computer architecture. In *EDBT*, pages 592–614, 1996.

[37] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for r-tree packing. Technical report, 1997.

[38] A. Mondal, M. Kitsuregawa, B. C. Ooi, and K. L. Tan. R-tree-based data migration and self-tuning strategies in shared-nothing spatial databases. In *9th ACM International Symposium on Advances in Geographic Information Systems*, pages 28–33, 2001.

[39] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.

[40] Y. Ohsawa and M. Sakauchi. A new tree type data structure with homogeneous nodes suitable for a very large spatial database. In *ICDE*, pages 296–303, 1990.

[41] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS*, pages 181–190, 1984.

[42] A. Papadopoulos and Y. Manolopoulos. Parallel bulk-loading of spatial data. *Parallel Computing*,

29(10):1419–1444, 2003.

[43] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *SIGMOD*, pages 17–31, 1985.

[44] B. Schnitzer and S. T. Leutenegger. Master-client r-trees: a new parallel r-tree architecture. In *SSDBM*, pages 68–77, 1999.

[45] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[46] Y. Tao, W. Lin, and X. Xiao. Minimal mapreduce algorithms. In *SIGMOD*, pages 529–540, 2013.

[47] P. Xu and S. Tirthapura. Optimality of clustering properties of space-filling curves. *ACM Transactions on Database Systems*, 39(2):10:1–10:27, 2014.

[48] S. You, J. Zhang, and L. Gruenwald. Parallel spatial query processing on gpus using r-trees. In *2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 23–31, 2013.

[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, 2010.

[50] R. Zhang, P. Kalnis, B. C. Ooi, and K. Tan. Generalized multidimensional data mapping and query processing. *ACM Transactions on Database Systems*, 30(3):661–697, 2005.

[51] R. Zhang, B. C. Ooi, and K.-L. Tan. Making the pyramid technique robust to query types and workloads. In *ICDE*, pages 313–324, 2004.

[52] R. Zhang, J. Qi, M. Stradling, and J. Huang. Towards a painless index for spatial objects. *ACM Transactions on Database Systems*, 39(3):19:1–19:42, 2014.