Noname manuscript No.
(will be inserted by the editor)

Instance Level Worst-case Query Bounds on R-trees

Yufei Tao · Yi Yang · Xiaocheng Hu · Cheng Sheng · Shuigeng Zhou

the date of receipt and acceptance should be inserted later

Abstract Even with its significant impacts in the database area, the R-tree is often criticized by its lack of good worst-case guarantees. For example, in *range search* (where we want to report all the data points in a query rectangle), it is known that, on adversely-designed datasets and queries, an R-tree can be as slow as a sequential scan that simply reads all the data points. Nevertheless, R-trees work so well on real data that they have been widely implemented in commercial systems. This stark contrast has caused long-term controversy between practitioners and theoreticians as to whether this structure deserves its fame.

This paper provides theoretical evidence that, somewhat surprisingly, R-trees are efficient in the *worst case* for range search on many real datasets. Given any integer K, we explain how to obtain an upper bound on the cost of answering *all* (i.e., infinitely many) range queries retrieving at most K objects. On practical data, the upper bound is only a fraction of the overhead of sequential scan (unless, apparently, K is at the same order as the dataset size). Our upper bounds are tight up to a constant factor, namely, they cannot be lowered by more than $O(1)$ times while still capturing the most expensive queries.

Our upper bounds can be calculated in constant time by remembering only 3 integers. These integers, in turn, are generated from only the leaf MBRs of an R-tree, but not the leaf nodes themselves. In practice, the internal nodes are often buffered in memory, so that the integers aforementioned can be efficiently maintained along with the data updates, and made available to a query optimizer at any time. Furthermore, our analytical framework introduces *instance-level query bound* as a new technique for evaluating the efficiency of heuristic structures in a theory-flavored manner (previously, experimentation was the dominant assessment method).

Y. Tao, X. Hu, and C. Sheng are with the Chinese University of Hong Kong. Y. Yang and S. Zhou are with Fudan University.

1 Introduction

R-trees [2,5,8,10,16] are considered by many to be one of the most important discoveries in spatial databases in the past three decades. We believe there are at least two main reasons. First, this access method works well in practice. Not only has this fact been demonstrated in a large number of papers, but it is also why R-trees are now commonly supported by commercial systems (e.g., Oracle, Microsoft SQL Server, DB2, etc.). Second, perhaps more importantly, the structure provides the *de facto* platform for studying algorithms for processing a great variety of spatial queries: *range search* [5], *nearest neighbor search* [7,15], *skyline* [12], to mention just a few. In particular, it has led to an interesting notion of *optimality*, namely, an algorithm is optimal if, for every query, it incurs the minimum cost among all the algorithms that use the *same* R-tree to solve that query. For example, the *best-first* algorithm of citehs99 and the *BBS* algorithm of [12] are well-known to be optimal in this sense for nearest neighbor and skyline queries, respectively.

R-trees are often lashed with criticism on their heuristic nature: they come with no attractive worst-case guarantees on query performance. Take range search as an example. In this problem, we are given a set P of 2d points (a.k.a. *objects*). Given a rectangle[1] q, a *range query* reports all the points of P covered by q. We want to build an R-tree on P so that queries can be answered efficiently. It has been shown [1,9] that: in the worst case, the query cost of an R-tree is as expensive as a naive *sequential scan* that simply reads the entire P, even when the query result is empty.

[1] All rectangles in this paper are axis-parallel.

The large gap between R-trees' good practical efficiency and terrible theoretical performance is intriguing enough to have generated long-term debates between theoreticians and practitioners on whether the structure deserves its fame. What is missing towards reconciling both sides is the lack of a convincing explanation on why the R-trees on real datasets appear to be efficient for every query experimented, even when they are compared to structures whose query time can be proved asymptotically optimal [1]. A bold question that remains unanswered is: perhaps those R-trees *are* efficient in the worst case after all?

Care must be taken to interpret "the worst query cost". If this simply refers to the cost of the most expensive query, then trivially it equals the total number of nodes in the R-tree – all nodes must be accessed by the query whose rectangle covers the entire data space. This, of course, is as meaningless as declaring that, in *one-dimensional* space, a B-tree on N points takes $O(N/B)$ I/Os to resolve a range query where B, called the *block capacity*, is the maximum number of objects that can be stored in a disk block. We all know that, to characterize a B-tree's behavior, we should instead bound its query cost in an *output-sensitive* manner as $O(\log_B N + K/B)$, where K is the *output size*, i.e., the number of points returned.

To pave the way for our discussion, let us look at $O(\log_B N + K/B)$ more closely. When given a specific B-tree (i.e., N being a fixed value), we can as well regard that complexity as a function of K, or formally

$$Q_{upper}(K) = c_1 + c_2 \cdot K/B \quad (1)$$

where c_1 and c_2 are positive values independent from K. That $O(\log_B N + K/B)$ is a worst-case query bound essentially says that, $Q_{upper}(K)$ upper bounds the cost of all queries whose output sizes are at most K (the number of such queries can be infinite). In other words, function $Q_{upper}(K)$ describes the relationship between the worst-case query time and the output size, for the underlying B-tree.

Our results. This paper focuses on 2d point data, for which R-trees have been very extensively deployed, especially in geographic information systems. We provide evidence that R-trees on many real datasets are indeed efficient for range search in the *worst case*. This is achieved by computing, for a given R-tree, a function $Q_{upper}(K)$ that upper bounds the cost of *all* range queries that report at most K objects. $Q_{upper}(K)$ is given in the form of Equation 1, with all constants made explicit (as opposed to constant hiding in big-O complexities). We will see that the value of $Q_{upper}(K)$ in practice is significantly smaller than the overhead of sequential scan, unless $K = \Omega(N)$ where all structures are (asymptotically) as expensive as sequential scan because $\Omega(N/B)$ I/Os is necessary just to report the qualifying objects.

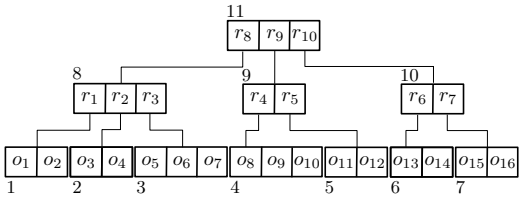
We also prove that our upper bound function $Q_{upper}(K)$ is tight up to a constant factor α, where α depends only on B and the minimum node utilization f (i.e., the smallest number of elements in a non-root node). In other words, it is impossible to reduce the upper bounds we claim by more than α times, while still ensuring that the resulting values correctly bound the cost of all queries. We coin the term α-*tight* for a function $Q_{upper}(K)$ with such a property. For a particular R-tree implementation (e.g., R*-trees, R$^+$-trees, Hilbert R-trees, etc.), our algorithm yields a $Q_{upper}(K)$ that is α-tight with the same α for all datasets, that is, the quality of our upper bounds is always guaranteed regardless of the dataset.

The computation of the proposed $Q_{upper}(K)$ can be accomplished using only the leaf MBRs of an R-tree, but not the leaf nodes themselves. In many systems, the internal levels of an R-tree are buffered in memory, because the number of all internal nodes is smaller than the dataset size N by a factor of nearly f, which is at the order of 100 in practice. We show that, in these systems, function $Q_{upper}(K)$ (or equivalently, values c_1 and c_2) can be efficiently maintained along with the data updates, and therefore, can be made available to a query optimizer at any time.

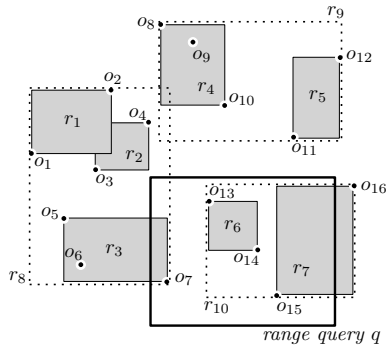
Of course, our results do not contradict the previous understanding that, when given adversely designed datasets and queries, the R-tree will have no advantage over sequential scan even for $K = 0$: in that case, our upper bound will automatically degenerate into $Q_{upper}(K) = cN/B$ for some constant $c > 0$. However, what is revealed by our results is that, dismissing the R-tree by citing its linear worst-case query cost is too pessimistic. When it comes to a practical scenario where the hardness of a dataset is unknown, one can apply our technique to examine how bad the query cost be possibly be on an off-the-shelf R-tree, before deciding whether specialized optimization needs to take place.

At the conceptual level, our technique introduces a new framework for evaluating *heuristic structures* in general. While previously experimentation has been the dominant assessment method, this paper shows how to draw an alternative theory-flavored conclusion. Unlike worst-case analysis that studies a structure's performance on *all* datasets, our analytical framework aims at its efficiency on an *individual* dataset, sharing the same spirit as *competitive analysis* [17]. However, different from competitive analysis that gives only the ratio from an algorithm's cost to the optimum, our goal is to derive the cost *explicitly* up to only a constant factor.

Paper organization. Section 2 surveys the previous work related to ours. Section 3 formally defines the concept of α-tight worst-case query bound. Then, Sections 4 and 5 explain our query bounds, prove their theoretical properties, and elaborate the algorithms for their computation. Section 6 experimentally evaluates the proposed techniques, and establishes the worst-case efficiency of R-trees on real



(a) The structure



(b) Data points and MBRs

Fig. 1 An R-tree

datasets. Finally, Section 7 concludes the paper with a summary of our findings.

2 R-tree

In this section, we review the knowledge of R-trees needed for our discussion. Since its invention [5], this structure has been improved in numerous ways, giving birth to several variants, most notably the R$^+$-tree [16], the R*-tree [2], the Hilbert R-tree [8], the STR-tree [10], and so on. Our description below applies to all these variants.

In an R-tree, all the data objects are stored at the leaf level, while an internal node u contains an entry referencing a distinct child node v of u. The entry is associated with the *minimum bounding rectangle* (MBR) of v, which is the smallest rectangle that encloses all the objects in the subtree of v. Figure 1 shows an R-tree on a set P of points $o_1, ..., o_{16}$. For instance, the MBR of leaf node 1 (Figure 1a), which stores objects o_1 and o_2, is rectangle r_1 (Figure 1b). As another example, the MBR of internal node 8 is r_8.

In general, every non-root node has $\Theta(B)$ objects/entries. An R-tree uses linear, namely, $O(N/B)$ space where N is the cardinality of the underlying dataset. Given a range query with search region q, the algorithm based on an R-tree accesses all and only the nodes whose MBRs intersect q. For example, in Figure 1, the query q demonstrated visits nodes 11, 8, 10, 3, 6, and 7.

Several studies [4,11,18,19] propose various *cost models* for estimating the number of I/Os performed by a range query. The estimates from those models are statistically accurate when the data distribution obeys certain assumptions. If this is not true, the estimates can arbitrarily deviate from the actual numbers, and therefore, are unsuitable for worst case analysis.

There exist linear-size structures that can process range queries with good worst-case guarantees. The external version of the kd-tree [3,13,14] supports a range query in $O(\sqrt{N/B} + K/B)$ I/Os, where K is the output size. The priority R-tree [1] achieves the same performance when objects are rectangles. This query bound is asymptotically optimal in the worst case [6,9]. Nevertheless, the query complexities of both the kd-tree and priority R-tree contain large hidden constants such that they are often outperformed by R*-trees in practice. The current paper (which concentrates on point data) will demonstrate this for the kd-tree in the experiments, whereas the experimentation of [1] points out the same phenomenon for the priority R-trees (whose superiority was established with contrived datasets in [1]). Neither the kd-tree nor the priority R-tree is implemented in any commercial DBMS.

3 Instance-level query bound

Conventionally, if a structure has query complexity, for example, $O(\log_B N + K/B)$, it is required that the bound should hold for all datasets – we say that such an upper bound is *universal*. Although prevailing in theoretical computer science, analysis based on universal upper bounds rules pessimistically that R-trees are as bad as sequential scan. To better capture the characteristics of an individual R-tree, we need to perform analysis on that specific instance.

Next, we formalize this intuition on range search. Let P be a set of points in two-dimensional space $\mathbb{R}^2$. Denote by $\mathcal{T}(P)$ the instance of an R-tree variant on P. It will be convenient to think of $\mathcal{T}$ as the construction algorithm of that variant. For example, if $\mathcal{T}$ represents the construction algorithm of R*-tree (or R$^+$-tree, ...), then $\mathcal{T}(P)$ is the R*-tree (or R$^+$-tree, ...) on P.

The query *cost* is defined as the number of leaf nodes in $\mathcal{T}(P)$ accessed by a query. We count only the leaf nodes because, in practice, the internal levels are often pinned in the buffer, so that visiting an internal node incurs no I/O. The next definition clarifies what is an instance-level query bound.

Definition 1 (INSTANCE QUERY BOUND) Let $\mathcal{T}(P)$ be an R-tree on a set P of N objects. A function $Q_{upper}(K)$ is an ***instance query bound*** of $\mathcal{T}(P)$ if the following holds for each integer $K \in [0, N]$: the cost of any range query with output size at most K is no more than $Q_{upper}(K)$.

Note that even if K is fixed, there can be infinitely many queries with output sizes at most K. The value of $Q_{upper}(K)$ must bound from above the cost of *all* those queries.

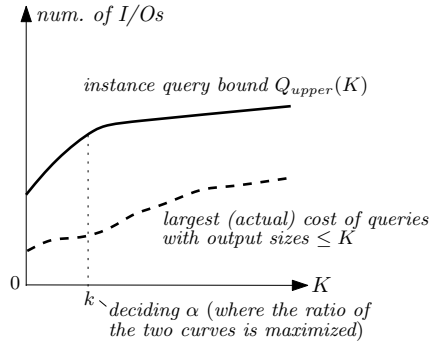


Fig. 2 Pictorial illustration of α-tightness

Many functions satisfy Definition 1 but not all of them are useful. For example, for a sufficiently large constant c, $Q_{upper}(K) = cN/B$ is a trivial instance query bound, even though it sheds no light on the quality of the given structure. The next definition is used to evaluate the quality of an instance query bound.

Definition 2 (α-TIGHT BOUND) Let α be a value at least 1. An instance query bound $Q_{upper}(K)$ is ***α-tight at $K = \tau$*** if for any $\alpha' > \alpha$,

$$\frac{1}{\alpha'} \cdot Q_{upper}(\tau)$$

no longer upper bounds the cost of all queries with output sizes at most τ. Function $Q_{upper}(K)$ is ***α-tight*** if it is α-tight at each integer $\tau \in [0, N]$.

We refer to α as the *tightness factor* of $Q_{upper}(K)$. The concept of α-tightness can be grasped from Figure 2, where (i) the dashed curve represents the cost of the *most expensive query* with output size at most K on an R-tree, and (ii) the solid curve represents an instance query bound $Q_{upper}(K)$ on the R-tree. As K ranges in its spectrum, the ratio of the two curves is *maximized* at the k shown in the figure. That maximum ratio is exactly the tightness factor α of $Q_{upper}(K)$. Put differently, if we reduce $Q_{upper}(K)$ by more than α times, the curve of the resulting $Q_{upper}(K)$ will be *strictly below* the dashed curve at any K.

Constant α. We consider only α that is a constant *independent from* N and K. The justification is that, for any function $f(N)$ monotonically increasing with N, an $f(N)$-tight instance query bound will be excessively loose when N is sufficiently large.

A note about universal bounds. A universal query complexity always implies a query bound at the instance level. For example, as mentioned in Section 2, a kd-tree on N objects answers a range query in $O(\sqrt{N/B}+K/B)$ I/Os. This means that, for some constant $c > 0$, function $Q_{upper}(K) = c\sqrt{N/B} + cK/B$ is an instance query bound. However, this bound is *not* necessarily $O(1)$-tight. The reason is that a universal bound is derived based on the *hardest* dataset. Consequently, on an easier dataset P, the structure's performance can be much better, such that the instance query bound adapted from a universal bound no longer accurately reflects the behavior of the structure's *instance* on P.

symbol	meaning
N	dataset size
B	block size
f	smallest number of entries in a non-root node
$K(q)$	output size of query q
$cost(q)$	cost of query q
$Q_{upper}(K)$	instance query bound
α	tightness ratio of an instance query bound
$upcross(S)$	upward crossing number of rectangle set S
$downcross(S)$	downward crossing number of rectangle set S

Table 1 Frequently used symbols

Generic and degenerate datasets. We say that P is *generic* if no two points of P share the same coordinate on any dimension (such P is often said to be in *general position*). In other words, any vertical or horizontal line in $\mathbb{R}^2$ passes at most one point in P. If this is not true, P is *degenerate* (however, P being a set implies the absence of duplicates, i.e., no two points coincide with each other). In Section 4, we will present our results for generic datasets. Section 5 will extend the discussion to degenerate datasets.

4 An instance query bound on Generic Data

We will explain how to produce an instance query bound on an R-tree $\mathcal{T}(P)$ where P is generic. Since our discussion will focus on the same P, $\mathcal{T}(P)$ will be abbreviated as $\mathcal{T}$ in this section. We will frequently use three topological relationships between rectangles. First, a rectangle r *intersects* another r' if $r \cap r' \neq \emptyset$. Second, r *covers* r' if $r' \subseteq r$. Finally, r *crosses* r' if r intersects, but is not covered by, r'.

4.1 The proposed bound

Denote by S the set of leaf MBRs in $\mathcal{T}$. If $\mathcal{T}$ indexes N objects, $|S| = O(N/B)$. Recall that f is the minimum number of elements in a non-root node of $\mathcal{T}$. It holds that:

$$f = \Omega(B) \tag{2}$$

where as mentioned before B, the *block capacity*, is the largest number of objects in a node. For a particular R-tree implementation, f/B is a constant in $(0, 1]$. For example, in an R*-tree [2], each non-root node is at least 40% full, namely, $f = 0.4B$.

Consider a range query with search rectangle q. When no ambiguity can arise, we will use q to refer to the query itself. Denote by $cost(q)$ the cost of using $\mathcal{T}$ to answer q, i.e., the number of rectangles in S intersecting q. We divide $cost(q)$ into two components:

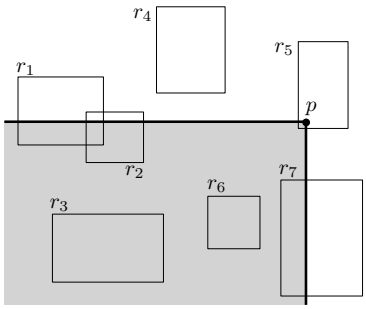


Fig. 3 Crossing number

- $cost_{edge}(q)$: the number of rectangles in S crossing q. Refer to these rectangles as the *crossing rectangles*.
- $cost_{inside}(q)$: the number of rectangles in S covered by q. Refer to them as the *inner rectangles*.

As no rectangle is counted by both $cost_{edge}(q)$ and $cost_{inside}(q)$, it holds that

$$cost(q) = cost_{edge}(q) + cost_{inside}(q). \quad (3)$$

For example, in Figure 1, $cost_{inside}(q) = 1$ due to r_6, whereas $cost_{edge}(q) = 2$ due to r_3 and r_7.

Let $K(q)$ be the output size of query q. Each inner rectangle in S represents a node that contributes at least f objects to the result, because all the (at least) f objects stored in the node must be covered by q. Hence:

$$f \cdot cost_{inside}(q) \leq K(q). \quad (4)$$

It is, however, difficult to capture $cost_{edge}(q)$ with $K(q)$. To see this, consider q being a vertical line which does not pass any point in P; thus, $K(q) = 0$ whereas $cost_{edge}(q)$ can be large. To analyze $cost_{edge}(q)$, we resort to a new concept:

Definition 3 (CROSSING NUMBER) Let S be a set of rectangles. Then:

- Given a point $p = (x, y)$ in $\mathbb{R}^2$, its ***downward quadrant number*** is the number of rectangles in S that cross the rectangle $(-\infty, x] \times (-\infty, y]$.
- The ***upward quadrant number*** of p is defined symmetrically with respect to the rectangle $[x, \infty) \times [y, \infty)$.
- The ***downward (upward) crossing number*** of S is the maximum of the downward (upward) quadrant numbers of all possible p.

Let $downquad(p, S)$ and $upquad(p, S)$ be the downward and upward quadrant numbers of p, respectively. Similarly, denote by $downcross(S)$ and $upcross(S)$ the downward and upward crossing numbers of S, respectively. By definition:

$$downcross(S) = \max_{p \in \mathbb{R}^2} downquad(p, S) \quad (5)$$

$$upcross(S) = \max_{p \in \mathbb{R}^2} upquad(p, S). \quad (6)$$

We give an example only about $downcross(S)$ because $upcross(S)$ is symmetric. Suppose that S consists of $r_1, ..., r_7$ as in Figure 3 (which are the leaf MBRs in Figure 1a). For the point p shown, we have $downquad(p, S) = 4$ which is the number of rectangles in S crossing the shadow area. It can be verified that p is a point in $\mathbb{R}^2$ maximizing the right hand side of Equation 5, i.e., $downcross(S) = 4$.

Lemma 1 *For any range query with search region q, $cost_{edge}(q) \leq downcross(S) + upcross(S)$.*

Proof Let p_1 (p_2) be the bottom-left (top-right) corner of q. Any rectangle counted by $cost_{edge}$ is also counted by the upward quadrant number of p_1 and/or the downward quadrant number of p_2. Hence, $cost_{edge}(q)$ is at most $upquad(p_1, S) + downquad(p_2, S)$, which by Equations 5 and 6 is at most $downcross(S) + upcross(S)$. □

We give the following function of K as an instance query bound:

$$\boxed{Q_{upper}(K) = downcross(S) + upcross(S) + K/f} \quad (7)$$

4.2 Tightness analysis

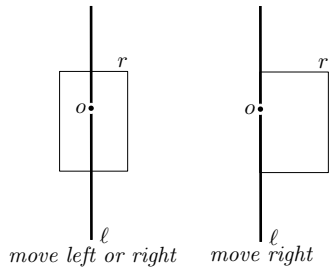
Next, we prove that the function in Equation 7 is an $O(1)$-tight instance query bound. We say that a range query is *empty* if it does not retrieve any object, i.e., its output size is 0. We first give a useful fact about such queries.

Lemma 2 *There is an empty query whose cost is at least $(downcross(S) + upcross(S))/4$.*

Proof Let ℓ be a vertical line that intersects the largest number of rectangles in S; denote the number as c_1. We argue that there is an empty query with cost c_1. In fact, if ℓ does not pass any data point in P, ℓ itself is the query we are looking for. Now assume that ℓ passes a point $o \in P$. Let r be the MBR of the node storing o. Figure 4 shows two cases depending on whether ℓ passes a boundary of r.

As P is generic, ℓ cannot pass any other point in P. This implies two facts. First, ℓ does not pass the left or right boundary of any rectangle in S except r. Otherwise, suppose ℓ passes the left boundary of a rectangle r' ($\neq r$) of S. Then, the leaf node whose MBR is r' must contain a data point whose x-coordinate is the same as o, resulting in a contradiction. The second fact implied is that ℓ can pass either the left or right boundary of r, but not both. Otherwise, r is a vertical segment, which is impossible for a generic P.

The above facts validate the following reasoning. If ℓ does not pass any boundary of r (Figure 4a), we can move it infinitesimally either to the left or right, without affecting the



(a) Case 1 (b) Case 2

Fig. 4 Proof of Lemma 2

set of rectangles in S it intersects. The ℓ after the movement corresponds to an empty query with cost c_1. If, on the other hand, ℓ passes, say, the left boundary of r (Figure 4b), we can still obtain such a query by moving ℓ infinitesimally to the right. Hence, an empty query with cost c_1 always exists.

Similarly, let c_2 be the maximum number of rectangles in S that are intersected by a horizontal line. By the same argument as above, there is at least an empty query with cost c_2. Without loss of generality, suppose $c_1 \geq c_2$. Next, we will show that c_1 is at least $(downcross(S) + upcross(S))/4$.

We will first prove that $downcross(S) \leq c_1 + c_2$. In fact, let $p = (x, y)$ be the point in $\mathbb{R}^2$ having the maximum $downquad(p, S)$, which determines $downcross(S)$. Due to the choice of c_1 (c_2), at most c_1 (c_2) rectangles in S intersect the right (upper) edge of the rectangle $(-\infty, x] \times (-\infty, y]$. As $downquad(p, S)$ counts each such rectangle *at most* once[2], $downcross(S) = downquad(p, S) \leq c_1 + c_2$. A symmetric argument shows that $upcross(S) \leq c_1 + c_2$. Now we have:

$$downcross(S) + upcross(S) \leq 2(c_1 + c_2) \leq 4c_1$$

which completes the proof. □

Now we prove the first main result of this paper.

Theorem 1 *The $Q_{upper}(K)$ in Equation 7 is an α-tight instance query bound of $\mathcal{T}$, where*

$$\alpha = \begin{cases} 2(\sqrt{1 + 2B/f} + 1) & \text{if } f \geq B/4 \\ 2B/f & \text{otherwise} \end{cases} \quad (8)$$

Proof The fact that $Q_{upper}(K)$ is an upper bound on any query with output size K follows directly from Equation 3, Inequality 4, and Lemma 1. The rest of the proof focuses on its tightness. Let $\beta \leq f/B$ be a parameter whose concrete value will be decided later. Set $c = downcross(S) + upcross(S)$.

[2] If the rectangle is covered by $(-\infty, x] \times (-\infty, y]$, it is not counted by $downquad(p, S)$.

We will first show that $Q_{upper}(\tau)$ is $(2/\beta)$-tight at any $\tau \in [\min\{N, c\beta B\}, N]$. In fact, a query q_1 with output size[3] τ accesses at least τ/B leaf nodes. Hence:

$$\begin{aligned} cost(q_1) &\geq \tau/B \\ (\text{from } \tau/B \geq c\beta) &\geq (\tau/B + c\beta)/2 \\ (\text{from } \beta \leq f/B) &\geq (\beta\tau/f + c\beta)/2 \\ &= \frac{1}{2/\beta} Q_{upper}(\tau). \end{aligned} \quad (9)$$

Thus, for any $\alpha' > 2/\beta$, the value $\frac{1}{\alpha'} Q_{upper}(\tau)$ will be *strictly* smaller than $cost(q_1)$, namely, $Q_{upper}(K)$ is $(2/\beta)$-tight at τ.

Next, we show that $Q_{upper}(\tau)$ is $(4 + 4\beta B/f)$-tight at any $\tau \in [0, \min\{N, c\beta B\})$. Let q_2 be the most expensive empty query. By Lemma 2, $cost(q_2) \geq c/4$. Hence:

$$\begin{aligned} cost(q_2) &\geq \frac{c(1 + \beta B/f)}{4(1 + \beta B/f)} \\ (\text{from } \tau < c\beta B) &> \frac{c + \tau/f}{4(1 + \beta B/f)} = \frac{Q_{upper}(\tau)}{4(1 + \beta B/f)} \end{aligned}$$

meaning that $Q_{upper}(K)$ is $(4 + 4\beta B/f)$-tight at $K = \tau$.

The earlier analysis suggests that the tightness factor of $Q_{upper}(K)$ is the maximum between $2/\beta$ and $4 + 4\beta B/f$. Our goal now is to choose a β to minimize the maximum. As $2/\beta$ decreases as β grows while $4 + 4\beta B/f$ increases, their maximum is minimized when they are equal. Solving $2/\beta = 4 + 4\beta B/f$ gives:

$$\beta = \frac{\sqrt{1 + 2B/f} - 1}{2B/f} \quad (10)$$

Recall that we have a constraint $\beta \leq f/B$ (as is needed in Inequality 9). When $f \geq B/4$, the β in Equation 10 always satisfies the constraint. Setting $\alpha = 2/\beta$ (which is also $4 + 4\beta B/f$) establishes the case $f \geq B/4$ in Equation 8, after some straightforward simplification.

When $f < B/4$, β cannot take the value in Equation 10. We instead set $\beta = f/B$, which makes $2/\beta = 2B/f$, and $4 + 4\beta B/f = 8$. This proves the other case of Equation 8, noticing that $2B/f > 8$. □

Because of Equation 2, the tightness factor α in Equation 8 is at most a constant. In particular, for R*-trees where $f = 0.4B$, we have $\alpha \approx 6.9$, whereas for bulkloaded R-trees (like STR-trees) where $f \approx B$, we have $\alpha \approx 5.5$.

4.3 Generating the instance query bound

We proceed to discuss how to generate function $Q_{upper}(K)$ of Equation 7 from an R-tree. The generation requires only

[3] Such a query always exists for a generic P. To see this, move a vertical sweep line ℓ rightward starting from $x = -\infty$, and stop as soon as τ points are on or to the left of ℓ. The swept region is a range query with size τ.

the set S of leaf MBRs. We note that $|S| \leq N/f$ where f (the minimum node utilization) is at the order of 100 in practice, that is, the size of S is only a fraction of N. Furthermore, S is readily available from the internal nodes pinned in the buffer. Hence, the gathering of S incurs no I/O, and neither does the generation of $Q_{upper}(K)$. We will show that the generation can be completed in $O(|S| \log |S|)$ CPU time.

Since f is known, the only terms to be decided in $Q_{upper}(K)$ are $downcross(S)$ and $upcross(S)$, as given by Equations 5 and 6, respectively. We will only explain how to compute $downcross(S)$, because a symmetric algorithm works for $upcross(S)$. In the sequel, let $n = |S|$.

From rectangles to points. We apply a transformation that allows us to attack instead a different problem on *points* (as opposed to rectangles). Construct a set A of $2n$ points as follows: for each rectangle r in S, add to A its bottom-left and top-right corners. Furthermore, associate the bottom-left corner with a value 1, and the top-right corner with -1. In other words, each point $a \in A$ carries a *weight*, denoted as $weight(a)$, that is either -1 or 1. Figure 5 illustrates the transformation for the rectangles in Figure 3. Here, A is the set of white dots, whose weights have been annotated next to them.

Given a point $p = (x, y)$ in $\mathbb{R}^2$, denote by $sum(p, A)$ the total weight of the points in A that are covered by the rectangle $(-\infty, x] \times (-\infty, y]$. For example, consider the point p in Figure 5, which is the same as in Figure 3. $sum(p, A) = 4$ because the shadow area covers six points with weights 1, and two points with weights -1. Notice that $sum(p, A)$ is equivalent to $downquad(p, S)$ (see Figure 3 again). The next lemma shows that this is true in general.

Lemma 3 *For any point $p \in \mathbb{R}^2$, it holds that $sum(p, A) = downquad(p, S)$.*

Proof Let r be a rectangle in S, and let a_1 and a_2 be the bottom-left and top-right corners of r, respectively. Let q be the rectangle $(-\infty, x] \times (-\infty, y]$. It is easy to see that, r crosses q if and only if a_1 is in q but a_2 is not. The lemma then follows. □

Therefore, instead of $downcross(S)$, we can compute $maxsum(A)$ defined as:

$$maxsum(A) = \max_{\forall p \in \mathbb{R}^2} sum(p, A). \quad (11)$$

It follows from Lemma 3 and Equation 5 that $maxsum(A) = downcross(S)$.

From the whole space to $|A|$ lines. Equation 11 still relies on all the points in $\mathbb{R}^2$. Next, we show that the equation can be calculated by focusing on much fewer points. Given a horizontal line ℓ, let $max\text{-}line\text{-}sum(\ell, A)$ be the maximum $sum(p, A)$ of all points p on ℓ, namely:

$$max\text{-}line\text{-}sum(\ell, A) = \max_{p \in \ell} sum(p, A). \quad (12)$$

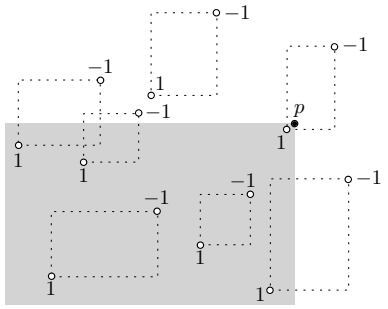


Fig. 5 Conversion for computing the downward crossing number of the set of rectangles in Figure 3

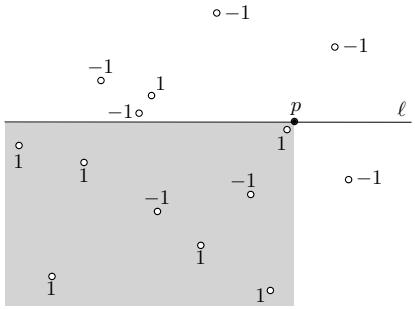


Fig. 6 Illustration of $max\text{-}line\text{-}sum(\ell, A)$

Consider, for example, Figure 6 where the white dots constitute A. For the line ℓ as shown, $max\text{-}line\text{-}sum(\ell, A) = 4$, as is decided by the $sum(p, A)$ of the black point p.

For each point $a \in A$, denote by $\ell(a)$ the horizontal line that passes a. It holds that:

$$maxsum(A) = \max_{\forall a \in A} max\text{-}line\text{-}sum(\ell(a), A). \quad (13)$$

To understand the above, imagine moving a point p up and down in the data space, and keep monitoring its $sum(p, A)$. It is clear that $sum(p, A)$ remains the same as long as it does not hit the $\ell(a)$ of any $a \in A$. Therefore, to compute $maxsum(A)$, we do not need to consider the $sum(p, A)$ of all points p in the data space; it suffices to concentrate on those p on the $|A|$ horizontal lines defined by the points in A. This is the implication from Equation 13.

4.4 Computing $maxsum(A)$

In this subsection, we describe a simple divide-and-conquer algorithm to compute $maxsum(A)$ based on Equation 13. Our strategy is to first calculate $max\text{-}line\text{-}sum(\ell(a), A)$ for *every* $a \in A$. Then, $maxsum(A)$ can be obtained by another scan of A.

Divide. Consider that the points in A have been sorted by their x-coordinates (otherwise, simply perform the sorting in advance). At the end of our algorithm, the points in A will have been sorted by y-coordinates. If A has only a single point a, set $max\text{-}line\text{-}sum(\ell(a), A) = 1$ if the weight of

a is 1; otherwise, $max\text{-}line\text{-}sum(\ell(a), A) = 0$. Next, we consider that $|A| \geq 2$.

Divide A by a vertical line into partitions A_1 and A_2 of the same size. That is, A_1 (A_2) includes all the points on the left (right) of the line. For each A_i, invoke our algorithm recursively. On return, by the earlier description, we know:

- for each $a \in A_i$, the value $max\text{-}line\text{-}sum(\ell(a), A_i)$ is ready;
- the points of A_i have been sorted by y-coordinates.

Conquer. We now merge A_1 and A_2 back into A, and in the meantime, acquire $max\text{-}line\text{-}sum(\ell(a), A)$ for each point $a \in A$. The merge processes the points in ascending order of y-coordinates. Equivalently, one can imagine lifting a horizontal sweep line ℓ from the bottom. A point a in A_1 or A_2 is processed when it is hit by ℓ. At all times, we maintain three integers:

- $max\text{-}line\text{-}sum(\ell, A_i)$ where $i = 1, 2$: this is as defined in Equation 12, but replacing A with A_i;
- $weightsum(\ell, A_1)$: the total weight of all the points in A_1 already swept, or formally:

$$weightsum(\ell, A_1) = \sum_{a \in A_1 \text{ on/below } \ell} weight(a).$$

These values can be updated easily whenever ℓ hits a point $a \in A_i$, namely, $\ell = \ell(a)$. Specifically, at this moment, we perform:

1. $max\text{-}line\text{-}sum(\ell, A_i) \leftarrow max\text{-}line\text{-}sum(\ell(a), A_i)$;
2. if $i = 1$, then $weightsum(\ell, A_i) \leftarrow weightsum(\ell, A_i) + weight(a)$.

Then, $max\text{-}line\text{-}sum(\ell(a), A)$ is determined as follows:

Lemma 4 *$max\text{-}line\text{-}sum(\ell(a), A)$ equals the maximum between $max\text{-}line\text{-}sum(\ell, A_1)$ and*

$$weightsum(\ell, A_1) + max\text{-}line\text{-}sum(\ell, A_2).$$

Proof Let p be the point on the sweep line ℓ that maximizes $sum(p, A)$. Hence, $sum(p, A) = max\text{-}line\text{-}sum(\ell, A)$. Let s be the split line that separates A_1 and A_2. If p is on the left of s, then p is also the point on ℓ that maximizes $sum(p, A_1)$. On the other hand, if p is on the right of s, then p is also the point on ℓ that maximizes $sum(p, A_2)$. Furthermore, in this case, $sum(p, A)$ equals $weightsum(\ell, A_1) + sum(p, A_2)$. The lemma thus follows. □

Example. Let us illustrate the merge phase using Figure 7. A_1 (A_2) includes all the points on the left (right) of the split line. Each black (white) point carries weight 1 (-1). Initially, $max\text{-}line\text{-}sum(\ell, A_1) = max\text{-}line\text{-}sum(\ell, A_2) = weightsum(A_1) = 0$. The first point hit by the sweep line ℓ is $a_1 \in A_2$, with $max\text{-}line\text{-}sum(\ell(a_1), A_2) = 1$ having been computed from recursion. At this moment, we set $max\text{-}line\text{-}sum(\ell, A_2) = 1$, after which

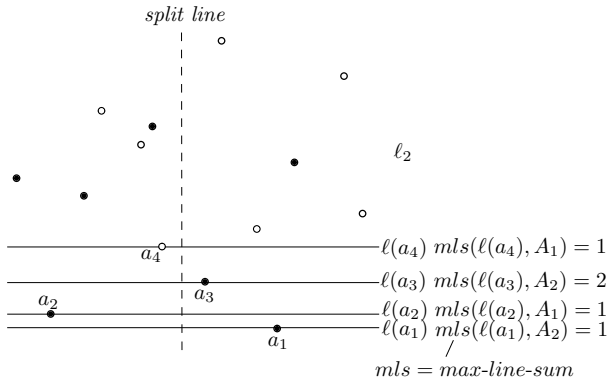


Fig. 7 Divide and conquer (black points have weight **1** and white points have weight $\mathbf{-1}$)

$max\text{-}line\text{-}sum(\ell(a_1), A)$ can be determined from Lemma 4 as $\max\{0, 0 + 1\} = 1$.

The second point swept by ℓ is $a_2 \in A_1$. The processing of this point sets $max\text{-}line\text{-}sum(\ell, A_1)$ to $max\text{-}line\text{-}sum(\ell(a_2), A_1) = 1$, increments $weightsum(A_1)$ to 1, and decides $max\text{-}line\text{-}sum(\ell(a_2), A) = \max\{1, 1 + 1\} = 2$ by Lemma 4. Likewise, the processing of the next point a_3 sets $max\text{-}line\text{-}sum(\ell, A_2)$ to $max\text{-}line\text{-}sum(\ell(a_3), A_2) = 2$, and calculates $max\text{-}line\text{-}sum(\ell(a_3), A) = \max\{1, 1 + 2\} = 3$.

The processing of a_4 is analogous, noting, however, that (since it has weight -1) it decreases $weightsum(A_1)$ back to 0. After setting $max\text{-}line\text{-}sum(\ell, A_1)$ to $max\text{-}line\text{-}sum(\ell(a_4), A_1) = 1$, we decide $max\text{-}line\text{-}sum(a_4, A)$ to be $\max\{1, 0 + 2\} = 2$. The rest of the algorithm proceeds in the same manner.

Running time. Let $F(x)$ be the CPU time of our algorithm on a dataset A with size $x = |A|$. The divide and conquer steps require $O(x)$ time. Hence, for $x > 1$:

$$F(x) = 2F(x/2) + O(x).$$

Combing the above with $F(1) = O(1)$ gives $F(x) = O(x \log x)$. We thus have arrived at the second main result of the paper:

Theorem 2 *Given a set S of leaf MBRs, we can generate the instance query bound $Q_{upper}(K)$ of Equation 7 in $O(|S| \log |S|)$ CPU time.*

4.5 Refreshing the instance query bound along with updates

Our instance query bound $Q_{upper}(K)$, which is a function described by only 3 integers, provides a query optimizer with an extra piece of information, i.e., a constant-tight upper bound on the query cost, once the query's selectivity has been estimated. If the dataset is static, the same $Q_{upper}(K)$

can be utilized forever. Next, we discuss how to maintain $Q_{upper}(K)$ in the presence of insertions and deletions.

A crucial observation is that, $Q_{upper}(K)$ does *not* change, as long as the set of leaf MBRs remains the same – even though the contents of some leaf nodes may have been altered. $Q_{upper}(K)$ needs to be re-computed only if one of the following occurs:

1. There is a node split/merge.
2. (For insertion) the new point does not fall inside any existing leaf MBR, or (for deletion) the deleted point lies on an edge of a leaf MBR.

Changes can occur to leaf MBRs only in the above scenarios.

Fortunately, these scenarios are not frequent as long as the data distribution remains stable with the updates (as is true in many applications). To see this, first note that, after a split/merge, a node needs to acquire/lose $\Omega(B)$ objects before the next split/merge can happen. For example, for R*-trees, a node must acquire at least $0.3B$ objects or lose at least $0.1B$ objects (this is quite pessimistic because, in general, the node may receive both insertions and deletions, so that its number of objects varies even slower). Therefore, splits and merges are infrequent, especially when leaf nodes have a similar chance to receive an update. Further, since a leaf node has at most 4 points on its MBR, the chance of deleting one of them is limited. Also, for a distribution to remain stable, insertions are essentially "filling up" the holes left by deletions. Thus, if most deletions are within leaf MBRs, then so are most of the insertions.

Remember that the re-generation of $Q_{upper}(K)$ is done completely in memory. Furthermore, our generation algorithm incurs only $O(|S| \log |S|)$ time – recall that $|S|$ is the number of leaf nodes, which is smaller than the dataset size by a factor of f (minimum node utilization, at the order of 100). Hence, the maintenance of $Q_{upper}(K)$ is not expected to incur expensive overhead, especially after cost amortization over an individual update.

5 An instance query bound on Degenerate Data

Now, we remove the constraint that P must be generic. In fact, given a degenerate P, our instance query bound on $\mathcal{T}(P)$ is exactly the same as Equation 7, and hence, can be generated in the same manner. The analysis, however, is slightly more complicated. We start with the lemma below which replaces Lemma 2.

Lemma 5 *There is an empty query whose cost is at least* $(downcross(S) + upcross(S))/8$.

Proof The argument is similar to the proof of Lemma 2, but extra care is needed to deal with the new case where many MBRs have their boundaries on the same vertical/horizontal line.

Let ℓ be the vertical line such that the number of rectangles in S crossing ℓ is the largest. Let R be the set of those rectangles, and $c_1 = |R|$. Note that for a degenerate P, an MBR can degenerate into a vertical segment, and hence, can be contained in ℓ. We will argue that there is an empty query with cost at least $c_1/2$. This is obvious if ℓ passes no point in P. Next, we focus on the opposite.

Let R_{left} (R_{right}) be the set of rectangles $r \in R$ such that ℓ crosses the left (right) edge of r. R_{left} and R_{right} are disjoint because R has no rectangle that degenerates into a vertical segment. Let R_{cut} be the set of rectangles of R that are in neither R_{left} nor R_{right}. Assume without loss of generality that $|R_{left}| \geq |R_{right}|$. It follows that $|R_{left} \cup R_{cut}| \geq |R|/2 = c_1/2$. We move ℓ infinitesimally to the right, after which ℓ intersects exactly the rectangles in $R_{left} \cup R_{cut}$, and does not pass any data point. We thus have found an empty query with cost at least $c_1/2$.

Define c_2 symmetrically with respect to horizontal lines. Suppose without loss of generality that $c_1 \geq c_2$. Slightly modifying the argument in the proof of Lemma 2, it is easy to show that $downcross(S) + upcross(S) \leq 4c_1$. The lemma then follows. □

Lemma 6 *For any* $\tau \in [1, N]$*, there is a range query whose output size is between* $\tau/2$ *and* τ.

Proof For $\tau = 1$, this is obviously true because each point in P can be a range query. Next, we assume $\tau \geq 2$. Consider first the scenario where there is a vertical line ℓ that crosses at least $\tau/2$ points in P. Then, from ℓ, we can cut out a segment that covers exactly $\tau/2$ points, and thus can serve as a query stated in the lemma.

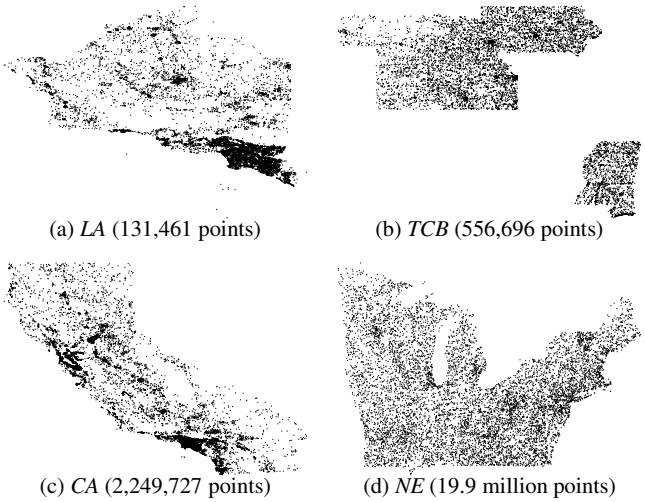
It remains to consider the case where every vertical line passes less than $\tau/2$ points. We construct a query as follows. Sweep a vertical line ℓ rightward from $x = -\infty$. In the meantime, monitor the number of points on or to the left of ℓ, and stop as soon as the number becomes no less than $\tau/2$. At this moment, the number must be less than τ; otherwise, there are at least $\tau/2$ points on the final ℓ, which as mentioned earlier cannot happen. We thus set the area having been swept by ℓ as the search region of a query, whose output size falls in $[\tau/2, \tau)$. □

Now we present the equivalent of Theorem 1:

Theorem 3 *For a degenerate* P *and* $f \geq 2$*, the* $Q_{upper}(K)$ *in Equation 7 is an* α*-tight instance query bound on* $\mathcal{T}(P)$ *where*

$$\alpha = \begin{cases} 4(\sqrt{1 + 2B/f} + 1) & \text{if } f \geq B/4 \\ 4B/f & \text{otherwise} \end{cases} \quad (14)$$

Proof Omitted because it is similar to the proof of Theorem 1, except that Lemmas 5 and 6 should be applied. □



(a) *LA* (131,461 points) (b) *TCB* (556,696 points)

(c) *CA* (2,249,727 points) (d) *NE* (19.9 million points)

Fig. 8 Datasets in our experiments

Note that the α in the above theorem doubles the one in Theorem 1, and is therefore also a constant.

6 Experiments

In this section, we will examine the worst-case efficiency of R-trees on real datasets by applying the techniques developed in earlier sections. We will also evaluate the cost of maintaining our instance query bounds along with data updates. All experiments were performed on a machine with a 3GHz CPU and Linux as the operating system.

Datasets. We deployed four real datasets:

- *LA* contains 131,461 points in the streets of Los Angeles.
- *TCB* is a collection of 556,696 points that represent places in Iowa, Kansas, Missouri and Nebraska.
- *CA* has 2,249,727 points in the streets of California.
- *NE* is a set of 19,909,725 point locations across the north-eastern region of the US.

The sizes of these datasets range from around 100k to nearly 20 million. Their distributions can be visualized in Figure 8. All datasets are publicly downloadable: *LA*, *TCB* and *CA* from the R-tree Portal (*www.rtreeportal.org*), and *NE* from the US Census Bureau (*www.census.gov/geo/www/tiger*).

R-trees and kd-tree. We examined two R-tree variants:

- the R*-tree [2], which is widely acknowledged as the most query-efficient among all the *incrementally-built* R-trees (i.e., constructing an R-tree by inserting each data point in turn).
- the STR-tree [10], which is a well-known *bulkloaded* R-tree, and is known to have better query efficiency than R-trees created by other bulkloading algorithms.

	seqscan cost	STR instance query bound	R* instance query bound
LA	386	$77 + K/339$	$103 + K/135$
TCB	$1,633$	$142 + K/339$	$207 + K/135$
CA	$6,598$	$288 + K/339$	$584 + K/135$
NE	$58,387$	$883 + K/339$	$1,328 + K/135$

Table 2 Instance query bounds of R-trees vs. the cost of sequential scan

Throughout the experiments, the size of a disk block is fixed to 4096 bytes, under which the block capacity B of an R-tree equals 339. As mentioned before, each leaf node in an R*-tree is at least 40% full, i.e., it contains at least $f = 135$ entries. On the other hand, an STR-tree is *packed*, meaning that each leaf node (except possibly one) contains the maximum number of entries, i.e., $f = B = 339$.

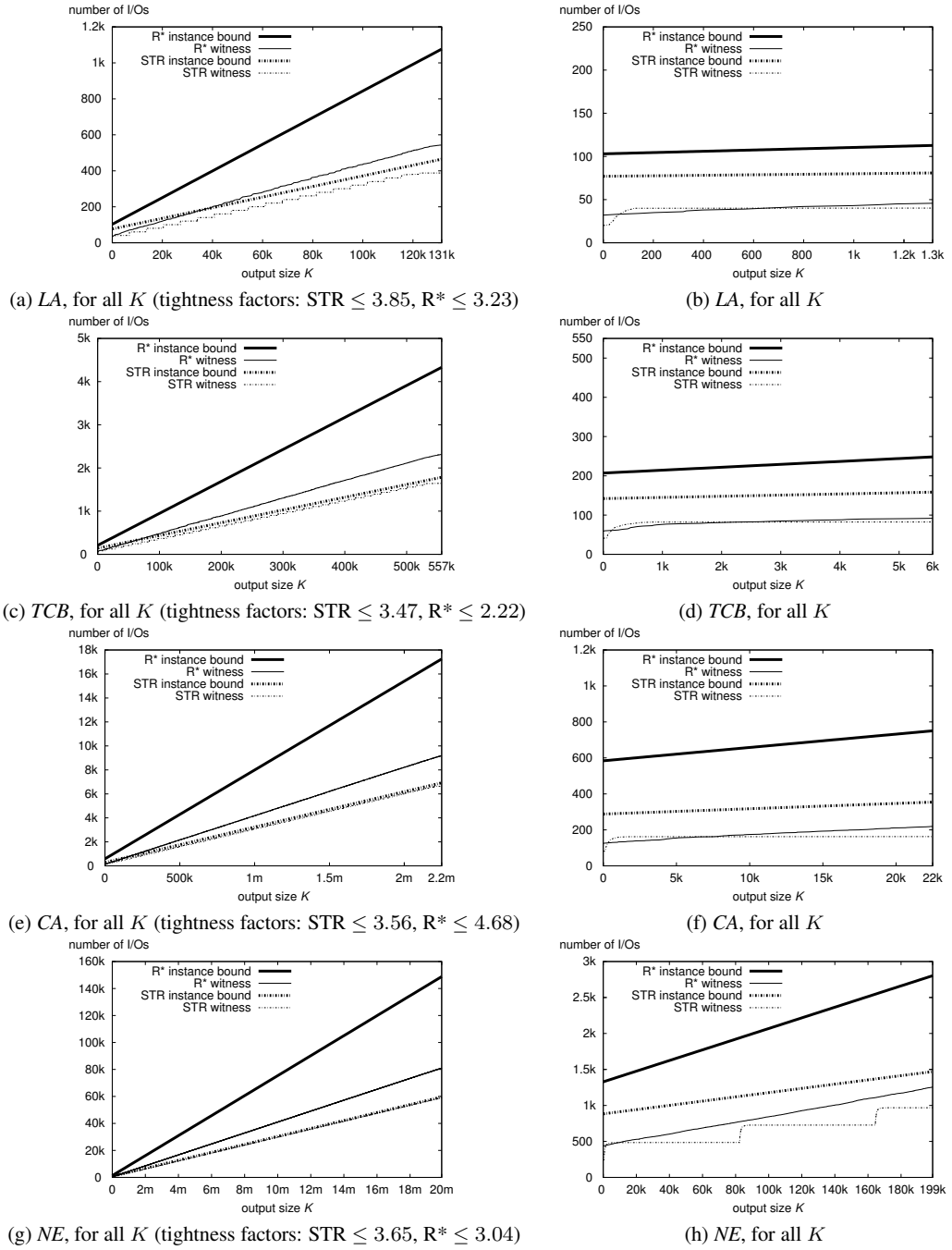
In several experiments, we will report the cost of sequential scan, abbreviated as *seqscan*, i.e., the naive solution that simply reads the entire dataset to answer a range query. The cost is equivalent to the smallest number of blocks required to store the underlying dataset.

Since worst-case query cost is used as the performance yardstick, it would be interesting to see how R-trees compare to a structure whose query efficiency has been theoretically proved asymptotically optimal. In the relevant experiments for this purpose, we used the kd-tree [3] as a theoretical representative, particularly, its external-memory variant in [13], which uses linear $O(N/B)$ space, and answers a range query optimally in $O(\sqrt{N/B} + K/B)$ I/Os, where N and K are the dataset and result sizes, respectively.

For each structure, all its internal nodes were pinned in the memory buffer. In all cases, the size of a buffer was always below $\frac{1}{230}$ of the underlying dataset's size (in general, a leaf node of an R*-tree is roughly 69% full [19]; hence, on average a leaf node stores $0.69B = 234$ objects).

Instance query bounds of STR- and R*-trees. The first set of experiments will show that, the R-trees on the datasets in Figure 8 significantly outperform *seqscan* in terms of worst-case query efficiency, unless the query result accounts for a significant portion of the dataset. We emphasize that our approach will *rigorously prove* the aforementioned fact, as opposed to empirical demonstration.

Table 2 lists the instance query bounds $Q_{upper}(K)$ determined (Equation 7) by our algorithm, for the STR- and R*-trees on various datasets, together with the cost of *seqscan*. Recall that, given an instance query bound (e.g., the STR bound $883 + K/339$ on *NE*), one can plug in any value of K to get an upper bound on the cost of *all* queries whose result sizes are at most K (e.g., any query outputting at most $K = 339$ points incurs at most 884 I/Os on the STR-tree of *NE*). It is thus clear that, on every dataset, the STR- and R*-trees entail only a fraction of *seqscan*'s overhead when K is small – this is especially true for K below 1% of the dataset



(a) *LA*, for all K (tightness factors: STR ≤ 3.85, R* ≤ 3.23)

(b) *LA*, for all K

(c) *TCB*, for all K (tightness factors: STR ≤ 3.47, R* ≤ 2.22)

(d) *TCB*, for all K

(e) *CA*, for all K (tightness factors: STR ≤ 3.56, R* ≤ 4.68)

(f) *CA*, for all K

(g) *NE*, for all K (tightness factors: STR ≤ 3.65, R* ≤ 3.04)

(h) *NE*, for all K

Fig. 9 Instance query bounds vs. witnessed query cost

cardinality. Also observe that this phenomenon is more evident when the dataset cardinality increases.

In a real system, a query optimizer would choose *seqscan*, unless the cost of an alternative index-driven plan is, say, 10 times lower (as *seqscan* benefits from sequential accesses). Our instance query bounds give a reliable threshold of K, below which an R-tree should *definitely* be selected. Take *NE* as an example. As long as $K \leq 1,679,745$ (8.4% of the cardinality of *NE*), the worst cost of the STR-tree, as calculated from Table 2, is at most $5,838$, namely, 10 times lower than *seqscan*.

Tightness of our instance query bounds. Recall that our instance query bounds are $O(1)$-tight, where the constant was proved in Section 4.2 to be at most 5.5 (6.9) for STR- (R*-) trees. We will demonstrate that the constant is lower in reality.

As explained in Definition 2 (and illustrated in Figure 2), to accurately measure the tightness factor α, we would need to know, for each $K \in [1, N]$, the maximum cost of all

queries reporting at most K objects; let us denote that cost as $Q_{max}(K)$. Since it is challenging to calculate $Q_{max}(K)$ accurately, we *disfavor* ourselves by presenting a *conservative* estimate $\hat{\alpha}$ that is *at least* as large as α. In other words, the actual tightness factor can only be *smaller* (i.e., better) than what we are to demonstrate next.

To obtain $\hat{\alpha}$, we resort to a *witness function* $Q_{witness}(K)$, where $Q_{witness}(K)$ is the cost of the most expensive query with output size at most K, among a set of generated queries (the generation will be elaborated shortly). Since $Q_{witness}(K)$ considers only a subset of queries considered by $Q_{max}(K)$, we know that

$$Q_{witness}(K) \leq Q_{max}(K) \quad (15)$$

holds at every K. Given an instance query bound $Q_{upper}(K)$, we decide its *conservative tightness factor* as:

$$\hat{\alpha} = \max_{\forall K} \frac{Q_{upper}(K)}{Q_{witness}(K)}$$

By Inequality 15, we can see $\hat{\alpha} \geq \alpha$ from:

$$\max_{\forall K} \frac{Q_{upper}(K)}{Q_{witness}(K)} \geq \max_{\forall K} \frac{Q_{upper}(K)}{Q_{max}(K)} = \alpha.$$

Next, we explain how to generate the set of queries used to obtain $Q_{witness}(K)$. We considered only *slab queries*, namely, the search region of a query is a vertical slab in the form of $[x_1, x_2] \times (-\infty, \infty)$. Once x_1 (i.e., the slab's left boundary) has been fixed, x_2 is determined by sliding the right boundary gradually until the slab covers exactly K objects. As for x_1, we set it to every possible x-coordinate in the underlying dataset. In other words, if the dataset has N objects, then N queries are created.

Figure 9a plots the $Q_{upper}(K)$ and $Q_{witness}(K)$ of the STR and R*-trees on dataset *LA* for the entire spectrum of K, while Figure 9b gives the two functions for the first 1% of the spectrum. The conservative tightness factors of the STR- and R*-trees are 3.85 and 3.23, respectively. Figures 9c-9h present the corresponding results on the other datasets. Note that, in all these figures, the two curves for STR are very close to each other. The tightness factors of the STR- and R*-trees are illustrated in figures' captions.

The next experiments examine how the tightness of our instance query bounds changes with respect to the dataset cardinality. Towards this purpose, for each dataset in Figure 8, we created 5 (sometimes miniature) replicas, by randomly selecting 20%, 40%, ..., 100% of its objects (apparently, the 100% replica is the same as the original dataset). For each replica, we measured the tightness factors of the corresponding STR- and R*-trees, as are presented in Table 3. The general observation is that, the tightness factor of an R-tree variant also depends on the data distribution, but appears to be less sensitive to the dataset size.

R-trees vs. an asymptotically optimal structure (namely, the kd-tree). A common understanding from the spatial and theory communities is that, an R-tree will lose to a theoretical structure proven worst-case efficient when they are compared in terms of the worst query cost (see, for example, [1]). Next, we revisit this understanding with the tools developed in this paper.

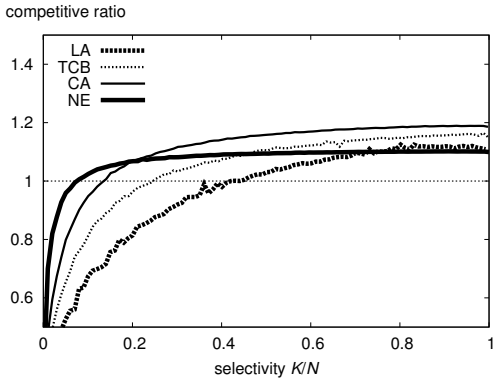
	20%	40%	60%	80%	100%
LA	3.67	4.00	3.94	3.84	3.85
TCB	3.64	3.54	3.58	3.56	3.47
CA	3.64	3.54	3.59	3.57	3.56
NE	3.66	3.66	3.64	3.64	3.65

(a) STR-trees

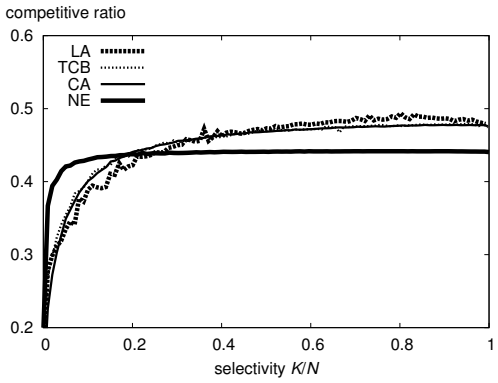
	20%	40%	60%	80%	100%
LA	3.59	3.95	3.82	3.84	3.23
TCB	2.87	3.43	2.27	3.56	2.22
CA	3.90	4.12	4.10	4.42	4.68
NE	3.11	3.07	3.04	3.15	3.04

(b) R*-trees

Table 3 Tightness factor as a function of dataset size



(a) STR-trees

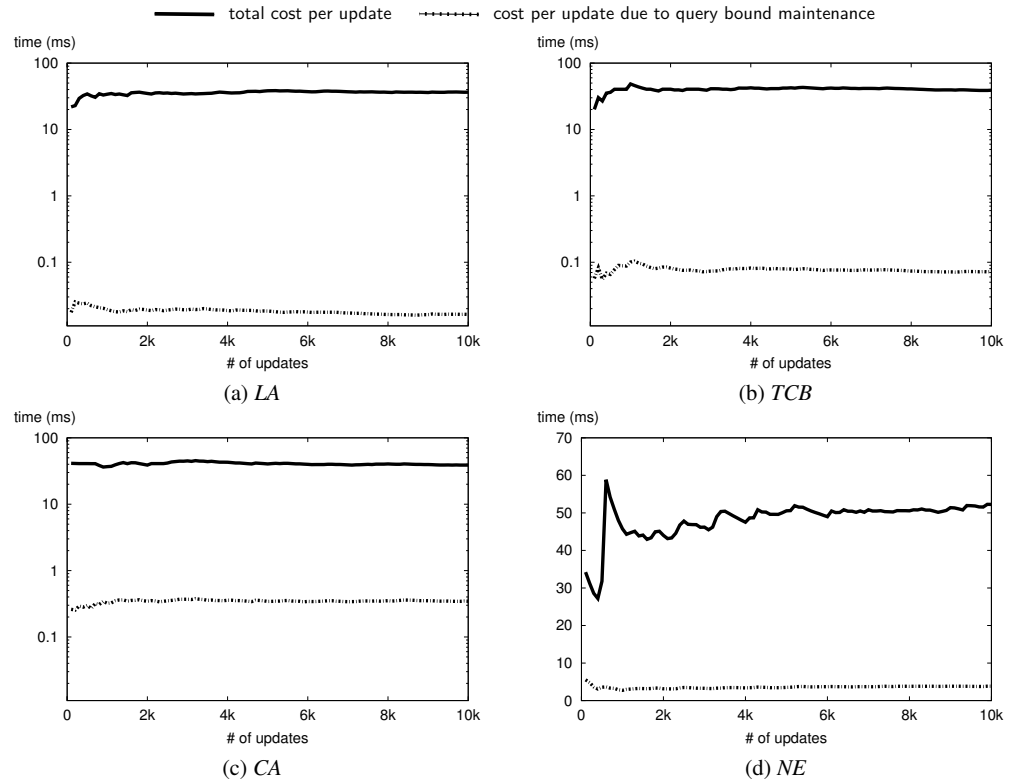


(b) R*-trees

Fig. 10 Worst-case competitive ratios of R-trees over kd-trees

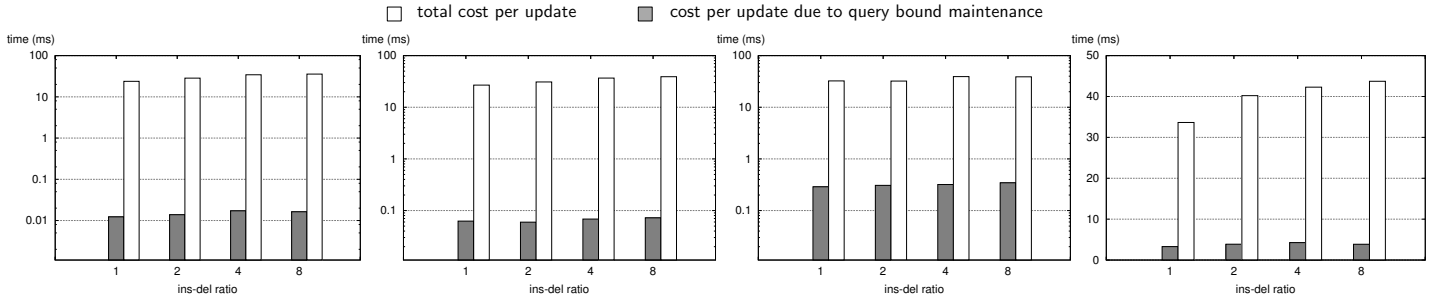
Our methodology is as follows. Consider an R-tree and a kd-tree on the same dataset. For a fixed K, let $Q_{max}(K)$ (as defined before) be the cost of the most expensive query with output size at most K on the R-tree. Define $Q^{kd}_{max}(K)$ similarly for the kd-tree. Then, the R-tree's *worst-case competitive ratio at* K equals:

$$r_{comp}(K) = Q^{kd}_{max}(K)/Q_{max}(K).$$



(a) *LA* (b) *TCB* (c) *CA* (d) *NE*

Fig. 11 Cost of maintaining an instance query bound vs. total cost of an update on an R*-tree (ins-del ratio = 8)



(a) *LA* (b) *TCB* (c) *CA* (d) *NE*

Fig. 12 Update cost vs. ins-del ratio

Because it is non-trivial to obtain the exact $Q_{max}(K)$ and $Q^{kd}_{max}(K)$, we favor the kd-tree by calculating an estimate $\hat{r}_{comp}(K)$ such that $\hat{r}_{comp}(K) \leq r_{comp}(K)$. In this way, if $\hat{r}_{comp}(K) > 1$, we know that $r_{comp}(K)$ must also be greater than 1, making it rigorously safe to claim that the R-tree is better.

We derive $\hat{r}_{comp}(K)$ as follows. For the kd-tree, we compute its witness function $Q^{kd}_{witness}(K)$ as described in the experiments of Figure 9. For the R-tree, we use directly its instance query bound $Q_{upper}(K)$ obtained by our technique. Then:

$$\hat{r}_{comp}(K) = Q^{kd}_{witness}(K)/Q_{upper}(K).$$

Since $Q^{kd}_{witness}(K) \leq Q^{kd}_{max}(K)$ yet $Q_{upper}(K) \geq Q_{max}(K)$, it follows that $\hat{r}_{comp}(K) \leq r_{comp}(K)$.

Figure 10a plots, for each dataset, the (conservative) competitive ratio $\hat{r}_{comp}(K)$ of the STR-tree (regarding the kd-tree on the same dataset) as a function of the *selectivity* K/N. We observed that the ratio always crosses 1 for large K. Furthermore, such a "crossing K" appears to decrease as the dataset size grows (remember that the cardinalities of *LA*, *TCB*, *CA* and *NE* are in ascending order). This observation implies a somewhat unexpected discovery: even in terms of the worst-case efficiency, *for a wide range of* K, the STR-tree can actually be better than a structure whose worst-case efficiency is theoretically proven to be optimal. Figure 10b demonstrates the corresponding results for R*-trees. Interestingly, the previous phenomenon was *not* observed: the R*-tree appears to be always worse than the kd-tree in terms of worst-case efficiency.

aspect ratio	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	1	2	4	8	16
STR, weakly data dependent	18.5	17.5	15.7	18.3	20.1	19.7	17.1	15.8	14.5
R*, weakly data dependent	36.0	36.9	35.0	36.4	43.0	43.2	40.5	38.6	33.8
STR, strongly data dependent	2.2	2.4	2.4	2.4	2.5	2.5	2.6	2.5	2.4
R*, strongly data dependent	4.1	4.1	4.2	4.2	4.2	4.3	4.3	4.1	3.9

(a) *LA*

aspect ratio	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	1	2	4	8	16
STR, weakly	15.9	15.7	15.4	16.8	16.7	15.3	12.2	10.5	10.0
R*, weakly	15.1	13.7	11.4	11.1	11.3	11.1	10.0	9.2	8.3
STR, strongly	2.9	3.0	3.1	3.3	3.4	3.5	3.6	3.6	3.7
R*, strongly	4.3	4.3	4.4	4.4	4.4	4.4	4.4	4.3	4.2

(b) *TCB*

aspect ratio	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	1	2	4	8	16
STR, weakly	11.3	17.3	21.8	22.2	23.3	23.4	22.5	21.5	16.9
R*, weakly	16.6	22.2	26.7	29.3	32.0	32.5	33.0	33.0	29.4
STR, strongly	4.6	4.8	4.9	5.0	5.0	5.0	5.1	5.1	4.9
R*, strongly	6.0	6.1	6.3	6.3	6.1	6.1	5.9	5.7	5.5

(c) *CA*

aspect ratio	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	1	2	4	8	16
STR, weakly	11.0	14.2	15.0	15.4	12.7	11.2	12.0	9.5	10.3
R*, weakly	12.8	17.0	16.5	18.4	19.5	17.3	17.6	17.7	25.8
STR, strongly	1.8	1.8	1.8	1.8	1.8	1.8	1.9	1.9	2.0
R*, strongly	2.6	2.7	2.7	2.7	2.7	2.7	2.7	2.8	2.8

(d) *NE*

Table 4 Average worst-actual ratios

Cost of updating instance query bounds. The next set of experiments is designed to assess the overhead of maintaining the proposed query bound, when an R-tree is updated with insertions and deletions. Towards this purpose, for each dataset in Figure 8, we first create an R*-tree using 90% of the objects. The remaining 10% objects are placed in an update *pool*. Then, we form a *workload* of 10,000 updates, where each update is either an insertion with probability p, or a deletion with probability $1 - p$. An insertion randomly removes an object from the pool, and inserts it into the R*-tree. Conversely, a deletion randomly deletes an object in the R*-tree, which is then added to the pool. STR-trees are not considered because they are inherently static.

The value of p controls the ratio between the numbers of insertions and deletions in the workload. We refer to the ratio as the *ins-del ratio*. For example, if $p = 1/2$, the workload has an expected ins-del ratio of 1, whereas if $p = 8/9$, then the ins-del ratio is expected to be 8.

Setting the ins-del ratio to 8, Figure 11a shows the results for dataset *LA*. The dotted curve gives the average (per-update) cost of maintaining our instance query bound (with respect to the updates already performed in the workload), as a function of the number of updates. For comparison, the solid curve gives the average cost of an entire update, again as a function of the number of updates. Figures 11b-11d demonstrate the results of the same experiment on *TCB*, *CA* and *NE*, respectively. Note that the y-axes of Figures 11a-11c are in log-scale.

Recall that the total cost of an update involves the time of performing I/Os. Each update requires at least 2 I/Os (one for reading, one for writing). The number of I/Os can be larger if overflows or underflows occur. It is evident that, in all cases, the overhead required to maintain our instance query bounds accounts for only a fraction of the total update time, confirming the discussion in Section 4.5.

To inspect the influence of the ins-del ratio, we doubled this ratio from 1 to 8, and for each ratio, measured the average (per-update) time of maintaining the instance query bound in handling an entire workload. The results are presented in Figure 12 for all datasets. As before, in each diagram, we included the average time of performing an entire update as a benchmark. It is clear from these results that the maintenance cost of our bounds is consistently low regardless of the ins-del ratio.

Average vs. worst-case performance. It is widely believed that R-trees' average query efficiency is much better than their worst case performance. The final set of experiments is designed to evaluate this belief. Our approach is as follows. Let $\mathcal{Q}$ be a set of queries with non-zero cost. For each query $q \in \mathcal{Q}$ with output size $K(q)$, define its *worst-actual ratio* as $Q_{upper}(K(q))/cost(q)$, where $cost(q)$ is the actual cost of answering q using the underlying R-tree, and (as before) $Q_{upper}(.)$ is our instance query bound on the R-tree. We define the *average worst-actual ratio* of $\mathcal{Q}$ as the average of the worst-actual ratios of all the queries in $\mathcal{Q}$. A large

average worst-actual ratio indicates that the R-tree's average efficiency on $\mathcal{Q}$ is much better than predicted in the worst case.

Next, we describe the generation of the query set $\mathcal{Q}$. For this purpose, we considered two factors: *aspect ratio* ρ and *centroid distribution*. The aspect ratio of a query rectangle q is the ratio between the lengths of q along the y- and x-dimensions, respectively. Fixing the area of each q to be 1% of the data space, we doubled the value of ρ from $\frac{1}{16}$ all the way to 16 (e.g., when $\rho = 1$, then each edge of q covers 10% of a dimension). For each specific ρ, we generated $\mathcal{Q}$ by two centroid distributions: *weakly* and *strongly* data-dependent, respectively. In the former, the centroid of each $q \in \mathcal{Q}$ distributes uniformly at random in the data space; however, if q does not access any leaf node (i.e., it has zero cost), it is re-generated until $cost(q)$ becomes non-zero. In the latter distribution, the centroid is placed at each point of the input dataset with the same likelihood. In any case, every $\mathcal{Q}$ consists of 10000 queries.

Table 4a demonstrates the results on dataset *LA*. Focusing on the STR-tree, the first row gives the average worst-actual ratios of query sets $\mathcal{Q}$ with various ρ, all of which were generated in the weakly data-dependent manner. The second row shows the same results for the R*-tree, whereas the next two rows are with respect to query sets under the strong data-dependent distribution. Tables 4b, 4c, and 4d present the corresponding results for datasets *TCB*, *CA*, and *NE*, respectively.

We make several observations from the above tables. First, when queries are weakly data dependent, the average efficiency of both the STR- and R*-trees is indeed much better than their worst-case performance, judging from the fact that all the average worst-actual ratios are quite large for such queries. Second, (again) for weakly data-dependent queries, the cost tends to be smaller when the query is more quadrate. Both observations, however, fade away when it comes to strongly data-dependent queries—the worst-actual ratios are considerably lower for such queries. The main reason behind this phenomenon is that these queries are less selective, i.e., they output more result points. Accordingly, both the predicted worst-case cost and their actual cost involve a heavy term proportional to K/B, which reduces the worst-actual ratio considerably. This fact also dampens the effect of the aspect ratio on the worst-actual ratio.

7 Conclusions

This paper has presented new progress towards answering the open question: how to prove that R-trees are (significantly) more efficient in practice than pessimistically predicted in theory? We have developed a set of theoretical tools for explaining this phenomenon in a rigorous manner, in contrast to all the previous heuristic attempts. Besides their merits in theory, our techniques also make a practical impact by allowing a query optimizer to impose a tight upper bound on the cost of any range query, once its selectivity has been estimated. Such information can be vital to query optimization, especially in scenarios where a critical limit on response time is demanded. We have also demonstrated that incorporation of our techniques into an existing system incurs only negligible extra overhead. We leave as an open problem how to derive constant-tight instance-level query bounds for rectangle data.

8 Acknowledgements

The research of Yufei Tao, Xiaocheng Hu, and Cheng Sheng was supported by grants GRF 4165/11, GRF 4164/12, and GRF 4168/13 from HKRGC. The research of Yi Yang and Shuigeng Zhou was supported by Research Innovation Program of Shanghai Municipal Education Committee under grant No. 13ZZ003. Finally, Yufei Tao would also like to acknowledge Discovery Project DP130104587 from the Australian Research Council.

References

1. L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 347–358, 2004.
2. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 322–331, 1990.
3. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM (CACM)*, 18(9):509–517, 1975.
4. C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 4–13, 1994.
5. A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 47–57, 1984.
6. J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *Journal of the ACM (JACM)*, 49(1):35–55, 2002.
7. G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.
8. I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of Very Large Data Bases (VLDB)*, pages 500–509, 1994.
9. K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 257–276, 1999.
10. S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 497–506, 1997.

11. B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 214–221, 1993.
12. D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems (TODS)*, 30(1):41–82, 2005.
13. O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. In *Proceedings of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 46–65, 2003.
14. J. T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 10–18, 1981.
15. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 71–79, 1995.
16. T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of Very Large Data Bases (VLDB)*, pages 507–518, 1987.
17. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM (CACM)*, 28(2):202–208, 1985.
18. Y. Tao and D. Papadias. Performance analysis of R*-trees with arbitrary node extents. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(6):653–668, 2004.
19. Y. Theodoridis and T. K. Sellis. A model for the prediction of R-tree performance. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 161–171, 1996.