Improving Spatial Data Processing by Clipping Minimum Bounding Boxes

Darius Šidlauskas¶, Sean Chester*, Eleni Tzirita Zacharatou¶, Anastasia Ailamaki¶

¶*École Polytechnique Fédérale de Lausanne, Switzerland*

* *Norwegian University of Science and Technology, Trondheim, Norway*

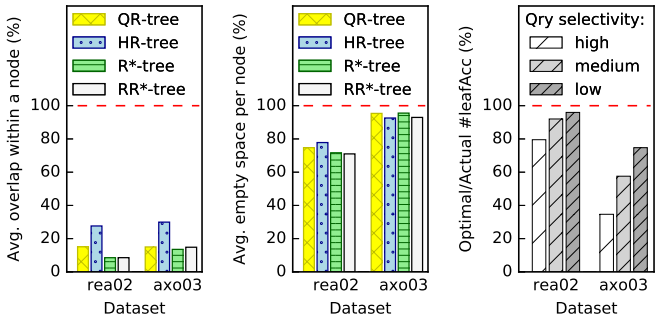
***Abstract*—The majority of spatial processing techniques rely heavily on approximating each group of spatial objects by their minimum bounding box (MBB). As each MBB is compact to store (requiring only two multi-dimensional points) and intersection tests between MBBs are cheap to execute, these approximations are used predominantly to perform the (initial) filtering step of spatial data processing. However, fitting (groups of) spatial objects into a rough box often results in a very poor approximation of the underlying data. The resulting MBBs contain a lot of "dead space"—fragments of bounded area that contain no actual objects—that can significantly reduce the filtering efficacy.**

This paper introduces the general concept of a *clipped* bounding box (CBB) that addresses the principal disadvantage of MBBs, their poor approximation of spatial objects. Essentially, a CBB "clips away" dead space from the corners of an MBB by storing only a few auxiliary points. On four popular R-tree implementations (a ubiquitous application of MBBs), we demonstrate how minor modifications to the query algorithm exploit auxiliary CBB points to avoid many unnecessary recursions into dead space. Extensive experiments show that clipped R-tree variants substantially reduce I/Os: e.g., by clipping the state-of-the-art revised R*-tree we can eliminate on average 19% of I/Os.

I. Introduction

Spatial data is growing at an alarming rate, prompting all major database system vendors to add spatial extensions that explicitly target spatial data analysis. From Oracle Spatial [1] and IBM Informix [2] to PostGIS [3] and HyPerSpace [4], the spatial index on which these extensions primarily rely is the heavily researched R-tree [5], and the principle component of the R-tree, all its variants, and in fact most spatial processing techniques, is a minimum bounding box (MBB). So, even minor improvements to MBBs can have broad impact.

Min. Bounding Box The MBB is the smallest axis-aligned rectangle that encloses its d-dimensional data. Being in a *conservative* class of approximations [6], it can represent any set of spatial objects (from simple points to complex volumetric shapes or other MBBs). Generally, a group of spatially-near objects are stored together (inside index nodes or buckets) and approximated by their MBB, offering three advantages: MBBs (i) are computed simply, with linear cost, (ii) are very compact to store, needing only two spatial points, and (iii) are very cheap to compare to each other for overlap/intersection. This is critical, as MBB intersection tests are the most dominant operation in spatial indexing: R-trees rely heavily on them during both the building and querying phases, while many other spatial tasks use them for filtering, e.g., traversing quadtrees [7] or performing spatial joins [8], [9].



(a) Overlap (b) Dead space (c) I/O optimality

Fig. 1: Performance of four R-tree variants.

Overlap, coverage, and "dead space" The quality of spatial data partitioning is classically measured by the *overlap* and *coverage* of the resultant MBBs. Poor overlap (i.e., much area spanned by multiple MBBs) decreases the filtering precision. This is particularly crucial for R-trees [10]: if a query rectangle intersects overlapping MBBs, the query must follow several paths in the tree. Reducing coverage (i.e., avoiding unnecessarily large MBB volume), however, is also important. Redundant coverage increases the likelihood that a query rectangle will intersect an MBB, independent of the likelihood of intersecting the constituent objects. How to minimize the overlap has been researched extensively in each new R-tree variant (R^+-tree [11], R*-tree [12], RR*-tree [13] to name a few).

In general, existing techniques minimize overlap quite well. In Figure 1a, we use two real-word datasets (described in Section V-B), including a novel and challenging brain axon dataset, to construct four popular disk-based R-tree variants (including the state-of-the-art, RR*-tree [13]). Indeed, on both datasets and all four variants, just 8–30% of the area of a node, averaged over all internal nodes, is overlapped by two or more of its children. However, Figure 1b is less encouraging. It instead measures what we call *dead space*, i.e., the percentage of the volume of a node that does not contain any objects. While one expects unnecessary coverage in higher dimensions due to the curse of dimensionality, we see a staggering amount, circa 74% and 94%, of dead space already in these $2d$ (`rea02`) and $3d$ (`axo03`) spatial datasets. This demonstrates the difficulty in bounding real objects such as road networks (`rea02`) and brain axons (`axo03`) with MBBs, even if they are easily separable.

Finally, Figure 1c illustrates that these hardly-overlapping

but mostly empty MBBs indeed have a negative effect on query performance. We query the (state-of-the-art) RR*-tree [13] under three settings of query selectivity that return a couple (high), about ten (medium), or roughly one hundred (low) objects. The plot reports the fraction of leaf node accesses—the major I/O bottleneck—that actually contribute to the query result (i.e., contain at least one spatial object within the query range). High/medium selectivity queries, common in spatial joins [8], [9], are particularly affected; the query intersects just dead space in 21% ($2d$) and 64% ($3d$) of the accessed leaf nodes. For the other three R-tree variants (not shown), the results are even more discouraging.

Our clipping proposal Certainly, we are not the first to observe the limitations of bounding boxes; various polygons [14], [15] and conics [16]–[19] have been proposed and compared [6], [20]. However, these are limited by: (a) the complexity of their representation, (b) the complexity of their intersection tests, and/or (c) the lower-bound on their dead space imposed by their convexity.

In contrast, we propose simple, non-convex polygons obtained by rectangularly clipping off the corners of MBBs (c.f., Figure 2 on the next page). Incidentally, the corners are the convergence points of the dimension-wise maxima/minima of the bounded objects and thus where much of the dead space is concentrated. Each clip requires only one d-dimensional point (and a d-bit flag) because the corner of the original MBB provides the opposing corner of the rectangular clipped area. Testing intersection with a clipped corner (we will show) is even cheaper than the preceding intersection test with the MBB. Finally, because the clipped corners are supplemental to the original MBBs, we can implement the proposal as a plugin-like addition for any R-tree variant: we store clip points in a small, auxiliary data structure, post-process MBBs at construction time, and minimally expand the query algorithm.

Clipping MBBs removes dead space and thereby achieves both classic objectives of spatial data partitioning: coverage is assuredly reduced by not representing dead space in the corners; overlap is potentially reduced by bounding the objects more tightly. Ideally we introduce very few clips that eliminate most dead space. We propose two means of generating clip points, based on the idea of Pareto optimality (i.e., skylines [21] in database literature), that trade-off complexity for pruning power. Our experiments with benchmark R-tree queries show that these proposals reduce leaf node accesses by 14% and 26%, while introducing a storage overhead of just 3.2% and 6.5%, respectively (averaged across seven datasets of 2–3 dimensions and four R-tree variants). Moreover, for two classic spatial join strategies, the Index Nested Loop Join and the Synchronised Tree Traversal, we eliminate 46% and 18% of I/Os, respectively (averaged across four R-tree variants).

Contributions and outline In this paper, we propose low-overhead improvements to minimum bounding boxes (MBBs) to improve their ability to represent complex, real spatial objects. While Section II details related work and Section VI concludes, our main contributions include:

- Introducing the general concept of *clipped bounding boxes* (CBBs) along with two particular instantiations of the concept (Section III);
- Demonstrating that our CBBs can be plugged into any R-tree variant with a small auxiliary data structure and minor modifications to the construction, query, and update algorithms (Section IV);
- Experimentally showing that with three-fold fewer corners, our CBBs can prune more area than the convex hull, while providing average I/O savings of 29, 29, 27, and 19 $\%$ when plugged into the QR-tree, HR-tree, R*-tree, and RR*-tree, respectively. In addition we are showing that our CBBs can save up to 53% of I/Os for the state-of-the-art spatial join strategies when plugged into R-tree variants (Section V).

II. Related Work

The R-tree family The R-tree [5] is a disk-based multi-dimensional index structure consisting of a hierarchy of minimum bounding boxes (MBBs) which recursively enclose data objects. Indexing and query processing with R-trees in spatial databases have received a lot of research attention [22] and several variants of the original approach have been proposed. To increase robustness against different data distributions, the R*-tree [12] incorporates an improved node split algorithm and the removal and reinsertion of spatial objects of an overflowing node. It employs multiple optimisation criteria at every split, attempting to minimise the dead space and the margin in each node as well as the overlap between nodes. The RR*-tree [13] introduces more adaptive optimisation strategies in order to further reduce I/O costs and enhance search performance.

Bulk loading can optimise the partitioning of data in the R-tree during initial construction. The HR-Tree [23] uses the Hilbert space filling curve to identify spatially close objects, while STR [24] recursively sorts the objects along each dimension for spatial proximity. To better handle extreme data, the PR-Tree [25] groups all objects with extreme coordinates in the same dimension in the same node.

All the above and many other access methods [26] operate on MBBs and thus our proposed clipping techniques can be applied orthogonally to boost their query performance.

Space-oriented partitioning Instead of grouping objects hierarchically based on their location, another family of spatial indexing methods splits space using hyperplanes into a set of disjoint partitions that are stored flatly [27] or in a hierarchical structure [7], [28]. To achieve good storage utilization, the hB-tree [29] does not require that nodes are split by hyperplanes; instead, it allows them to represent hyperrectangular regions with hyperrectangular holes. Note that, similarly to CBBs, the hole can be located at the corner of the node. However, in such cases, the corner region is not a dead area, but is in fact *guaranteed to contain data objects*. By definition, space-oriented partitions do not minimally bound the enclosed data objects and therefore contain dead space.

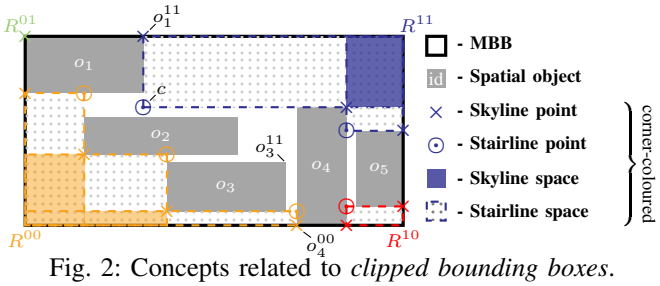


Fig. 2: Concepts related to *clipped bounding boxes*.

Bounding objects Instead of using minimum bounding boxes to represent a collection of objects, other geometries have been proposed in the past. The Sphere-tree [18] is a hierarchical structure similar to the R-tree with the exception that it uses minimum bounding spheres. The Sphere-tree requires less storage space than the R-tree but the computation of the minimum bounding sphere is more expensive [30]. The SS-tree [17] indexes multidimensional points and employs bounding spheres having as center the centroid of the underlying points for the region shape. The SR-tree [16] integrates both bounding spheres and bounding boxes to reduce the volume that an index node occupies and increase search efficiency. However, the SR-tree has lower fanout (its nodes contain both a sphere and a box) and higher creation cost compared to the SS-tree. The eR-tree [19] is a variant of the R-tree that employs minimum volume covering ellipsoids. Ellipsoids cover less dead space on sparse clustered data but their advantage is suppressed on dense data. Moreover, they cover more dead space than polygonal alternatives [6], [20]. The P-tree [14], another generalization of the R-Tree, uses a hierarchy of polygon containers that are mapped into rectangles of a higher dimension. The use of additional hyperplanes results in a better approximation, but it increases the size of the entries (i.e., reduces the fanout of interior nodes). The Cell Tree [15] is a hierarchy of nested convex polyhedra that are subspaces of a binary space partitioning and aims to handle objects of arbitrary shape. In contrast, our work is not proposing an alternative geometry, as MBBs have been tremendously successful in practice. Instead, we extend MBBs using clip points that adhere to the same basic, rectilinear concept.

Recently, [31] tackled the same challenge, "over-coverage" of MBBs, focusing solely on streaming observational data where a group of (successive) points can be represented as line segments. Our proposal is more general, supports points (lines/planes) as well as volumetric spatial objects, and does not imply any restrictions on the data (e.g., value-continuity of observations).

III. Eliminating dead space in MBBs

Figure 2 gives an example that runs over the next two sections. It depicts a set $O = \{o_1, \ldots, o_5\}$ of five (gray) spatial objects and the (black) box that minimally bounds them. It is clear that most of the space inside this MBB is not covered by an object, but is instead "dead space." We show in this section how to eliminate a lot of that dead space using only a few well-chosen points and bitmasks. These points "clip away" corners of the MBB, producing what we call a clipped bounding box. Section III-A introduces the general concept and Sections III-B–III-C define two specific implementations.

A. The clipped bounding box (CBB)

High-level concept Intuitively, a lot of the dead space in an MBB is tucked into the corners, where the most extreme values of all the objects converge. Incidentally, the rectangular area A adjacent to a corner is also the cheapest to represent: A, like any box, is defined by two points, one of which is the corner point; so, a single d-dimensional point and a d-bit flag to identify the targeted corner are sufficient to fully characterise A. Our proposed *clipped bounding box* (CBB) augments an MBB with pairs of additional points and corner flags to *clip away* excess area near the corners of the MBB. The reduction in dead space can be quite profound. In Figure 2 for example, one can store the point c with the bitmask $b = 11$ to represent the rectangle $\langle c, R^{11} \rangle$, which contains *no* objects. If this region was not part of the MBB, one would have an equally correct, but more representative, bounding object. Moreover, the cost of intersecting a query region Q with this clipped rectangle is low: we need only compare Q to the MBB (which, anyway, would be necessary), and then to point c.

In general, there are many choices for c and we need not select only one. Choosing the right points is, unsurprisingly, non-trivial and thus the subject of Sections III-B–III-C. First, however, we formalise this high-level intuition by introducing notation and definitions.

Notation An MBB in d dimensions is a hyperrectangle, which we denote by $R = \langle l, u \rangle$. The two points defining R, l and u, represent the minimum and maximum extent of R, respectively. We express coordinates of a point $p = (p[1], \ldots, p[d])$ and the bits of a bitmask $b = \langle b[1] \ldots b[d] \rangle$ in array notation. The *minimum bounding box* (MBB) of a set of m objects $O = \{o_1, \ldots, o_m\}$ is the smallest possible rectilinear box $R = \langle l, u \rangle$ that contains all objects in O; i.e., for each dimension i, $l[i] = \min_{o_j \in O} o_j[i]$ and $u[i] = \max_{o_j \in O} o_j[i]$. We frequently refer to specific corners of bounding boxes (hyperrectangles). For hyperrectangle R a unique superscripted bitmask b specifies the corner of interest: a set bit $b[i]$ indicates that the corner maximises the i'th dimension; i.e.,

$$R^b[i] = \begin{cases} u[i], & \text{if } b[i] = 1 \\ l[i], & \text{if } b[i] = 0. \end{cases}$$

The four corners of R, the MBB in Figure 2, are labeled with this notation. For example, the top-left corner is R^{01} and the top-right corner is R^{11}. Similarly, the top-right corner of object o_1 is denoted o_1^{11} and the bottom-left corner of object o_4 is denoted o_4^{00}.

Formalisation In general, an MBB is an imperfect approximation of a set of objects O, which a CBB improves. The extra, empty space that an MBB uses to bound O we call *dead space* and define as the part of R not occupied by any o_i of O (Definition 1):

Definition 1 (Dead space)**.** *Let R be the MBB of objects $O = \{o_1, \ldots, o_m\}$. The* dead space *of R, denoted $\dagger(R)$, is:*

$$\dagger(R) = \{p \in R : \forall o_i \in O, p \notin o_i\}.$$

A CBB augments an MBB with extra points that we call *clip points* (Definition 2). A clip point is a pair consisting of a point $p \in R$ and an orientation mask b and has the property that no object $o_i \in O$ occupies the space between p and its relevant corner R^b. (Observe that the space between p and R^b is exactly the MBB of the two points $\{p, R^b\}$.)

Definition 2 (Clip point)**.** *Let R be the MBB of objects $O = \{o_1 \ldots, o_m\}$ and b be a bitmask of length d. We say that $\langle p \in R, b\rangle$ is a* clip point *iff the area between p and R^b is entirely dead space; i.e., if R' denotes the MBB of $\{p, R^b\}$, then p is a clip point iff:*

$$\forall q \in R', \forall o_i \in O, q \notin o_i.$$

For example, $\langle c, 11\rangle$ is a clip point in Figure 2, because the area enclosed by the dashed blue lines is empty. On the other hand, $\langle o_4^{00}, 11\rangle$ is *not* a clip point, because it would clip away objects o_4 and o_5: the area between o_4^{00} and the top-right corner, R^{11}, is not entirely dead space.

We denote the volume that clip point $\langle p, b\rangle$ clips away by $\mathrm{Vol}_R(\langle p, b\rangle)$. In general, we clip away several parts of an MBB using a set of $\langle p_i, b_i\rangle$ pairs, but take care not to double-count regions clipped away by multiple clip points. That is to say, given a set of clip points $P = \{\langle p_i, b_i\rangle\}$, $\mathrm{Vol}_R(P) = \bigcup_{\langle p_i, b_i\rangle \in P} \mathrm{Vol}_R(\langle p_i, b_i\rangle)$.

This leads us to our core concept, a *clipped bounding box* (CBB), that we informally introduced at the beginning of this subsection and that we formally define in Definition 3:

Definition 3 (Clipped Bounding Box (CBB))**.** *Given a set of objects O, a* clipped bounding box *is a pair $\langle R, P\rangle$, where R is the MBB of O and P is a set of clip points in R.*

Naturally, a CBB, $\langle R, P\rangle$, is a better approximation of O than another CBB, $\langle R, P'\rangle$, if it retains less volume; i.e., $\mathrm{Vol}_R(P) > \mathrm{Vol}_R(P')$. While a clip point introduces very little overhead, a large set of clip points is cumbersome. Thus, we only want to select $\leq k$ of the clip points that we propose in the following subsections, while still maximising $\mathrm{Vol}_R(P)$.

B. Object-situated clip points

High-level concept Given an MBB R, we can obtain good clip points quickly by taking them from the objects O bounded by R. Consider Figure 2 again; (a discretisation of) the set of possible clip points is depicted by small, gray dots. For the most part, the best clip points (i.e., the dots farthest from their respective corners) lie on the outer surface of some object o_i. This is intuitive: the dead space arises specifically because the MBB corner is too far from the objects, so clipping R with a rectangle that borders an object o_i will naturally improve the approximation of O.

With respect to a corner R^b, we do not consider all (infinitely many) points on the surface of each object $o_i \in O$, but rather just its closest corner, o_i^b. This is most likely to be a valid clip point. In fact, if $\langle o_i^b, R^b\rangle$ is not a clip point, then *no* point in o_i can be a clip point with respect to R^b. Considering, for example, corner R^{11} in Figure 2, we see that $\langle o_3^{11}, R^{11}\rangle$ is not a clip point (it would clip away part of o_4 and o_5), so the entirety of o_3 would also clip away part of o_4 and o_5 and therefore similarly not be clip points.

We also do not necessarily consider all objects $o_i \in O$, because we recognise that, for corner R^b, the subset of $\{\langle o_i^b, R^b\rangle\}$ that are clip points is precisely the well-studied concept of a skyline [21], computed over $\{o_i^b\}$. Thus, the idea for our object-based CBB is to compute for each bitmask b the skyline of the M object corners $\{o_i^b\}$ and pick from those the few points that clip away the most dead area. Thus the clip points are all actually represented in the set of underlying objects, O. To formalise this intuitive description, we will state the definition of a skyline as it relates to this context.

Oriented skyline of objects The skyline [21] is based on the concept of *dominance*, which is highly related to our notion of clip points. A point p dominates a distinct point q with respect to b if it is at least as close to R^b as q on each dimension independently. More precisely, let $b_{p \odot q}$ denote a bitmask with bit i set iff $p[i] \odot q[i]$. Then we have Definition 4:

Definition 4 (Dominance)**.** *Given points p and q, and bitmask b, p dominates q w.r.t. b, denoted $p \prec_b q$, iff:*

$$(b_{p \leq q} \;\&\; {\sim}b = b_{p \leq q}) \wedge (b_{q \leq p} \;\&\; b = b_{q \leq p}) \wedge b_{p \neq q} \neq 0.$$

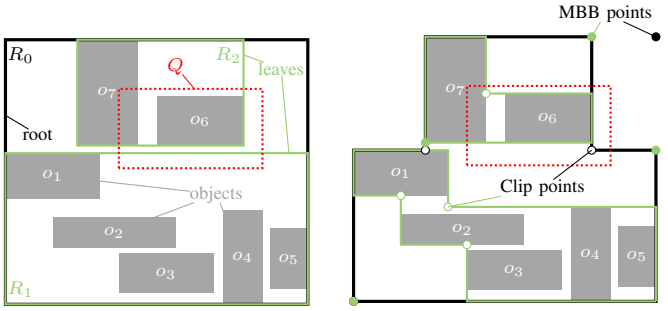
For example, given $b = 00$ in Figure 2, $o_4^{00} \prec_b o_5^{00}$ because it is closer to R^{00} in both the x and the y direction. In the context of MBBs, one could express dominance equivalently by letting R' denote the MBB of $\{q, R^b\}$ and stating that $p \prec_b q$ iff $p \in R'$. Using the same example, observe that the point o_4^{00} indeed lies in the MBB created by the point it dominates, o_5^{00}, and R^{00}. The *oriented skyline* of a set of points P, given orientation mask b, is simply the subset of points $p \in P$ not dominated by any other point $q \in P$:

Definition 5 (Oriented skyline)**.** *Given point set P and orientation mask b, the* skyline $\mathcal{S}_b(P)$ *is:*

$$\mathcal{S}_b(P) = \{p \in P : \nexists q \in P, q \prec_b p\}.$$

Considering an MBB R and a set of object corner points $P = \{o_i^b\}$, a pair $\langle p \in P, b\rangle$ is clearly a clip point iff $p \in \mathcal{S}_b(P)$. If a point $p \notin \mathcal{S}_b(P)$ is dominated by some other point $q \in P$, then q lies in the MBB of $\{p, R^b\}$ which implies that $\langle p, b\rangle$ would clip away the object from which q was derived. On the other hand, if $p \in \mathcal{S}_b(P)$, then no other point $q \in P$ lies in the MBB R' of $\{p, R^b\}$. Since P contains the closest point to R^b from every object $o_i \in O$, the entirety of $\langle p, R^b\rangle$ must contain dead space. Thus, the clip points in $\{\langle o_i^b, b\rangle\}$ are in one-to-one correspondence with $\mathcal{S}_b(\{o_i^b\})$.

We see this in Figure 2. Considering corner $b = 00$, for example, we obtain a skyline of $\{o_1^{00}, o_2^{00}, o_3^{00}, o_4^{00}\}$. Point o_5^{00} is dominated by both o_3^{00} and o_4^{00}; meanwhile the clip point $\langle o_5^{00}, b\rangle$ would clip away part of o_3 and o_4.



(a) A two-level R-tree (b) A clipped R-tree

Fig. 3: An example of an R-tree before (a) and after (b) clipping, given 7 objects, o_1–o_7 and a range query, Q.

C. Point-spliced clip points

High-level concept We can find more aggressive clip points by *splicing* the skyline points proposed in Section III-B. Recall the possible clip points in Figure 2. Skyline point o_4^{11} clips away a lot of dead space relative to R^{11}, but the point c that combines the y-coordinate of o_4^{11} with the x-coordinate of o_1^{11} clips away significantly more dead space. In fact, c clips away the most dead space (of those that could form valid clip points). c is not an arbitrary point in $\mathbb{R}^2$, but a combination of the coordinates of o_1^{11} and o_4^{11}: this *splicing* provides a generative mechanism of strong clip points that may not lie on any object at all, and comprises our second instantiation of CBBs. It clips away much more dead space than our first CBB proposal, but requires an expensive extra processing step.

Stairline points We define *stairline points*, which, intuitively, are the points "between" skyline points, farthest from a corner R^b of the MBB. To find them, we introduce the *splice point* concept, which mixes the coordinates of source points p, q (thereby still being adjacent to child MBBs):

Definition 6 (Splice point)**.** *Given two points p and q with MBB R, their* splice point *with respect to b is $\text{⋈}_b(p,q) \equiv R^b$.*

Stated alternatively, $\text{⋈}_b(p,q)[i]$ has value $\max(p[i], q[i])$ if $b[i]$ is set; otherwise, it takes the value $\min(p[i], q[i])$. For example, c in Figure 2 is equal to $\text{⋈}_{00}(o_1^{11}, o_4^{11})$, i.e., takes the smallest x and y values from its source points o_1^{11} and o_4^{11}, as the bitmask $b = 00$ specifies to minimise both dimensions.

We must be careful when splicing points to not clip away occupied area, i.e., to ensure the generated clip points are "valid." Producing valid clip points in $2d$ is straight-forward: we can *totally order* all skyline points by x and then splice each consecutive pair. However, it is non-trivial in higher dimensions to efficiently extract all sets of neighbouring points.

Thus, we propose an unfortunately-cubic algorithm that is still practically reasonable given the small input sets ($< M$). We generate splice points using bitmask $\sim b$ from every pair of skyline points, some of which are invalid; to ascertain validity, we check that no other skyline point (and thus child MBB) is in the space that this splice point would clip away.

Definition 7 (Oriented stairline)**.** *Given a set of points P and a bitmask b, the* oriented stairline *is the subset of splice points,*

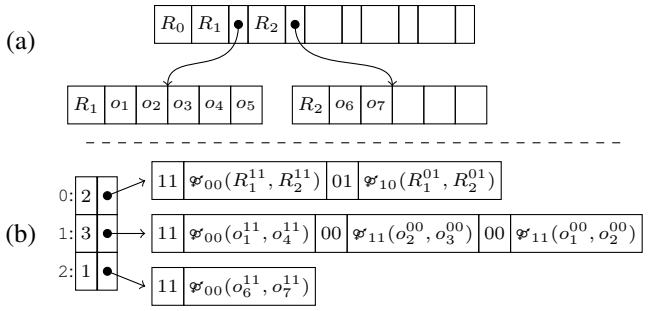


Fig. 4: The physical layout of the R-tree from Figure 3a (a) and the auxiliary structure (b) of clip points introduced in Figure 3b. The auxiliary table is indexed by MBB id and gives the number of and pointer to the clip points for that MBB. The bitmask of each (in this case $2d$) clip point is followed by the two coordinate values.

$\{\text{⋈}_{\sim b}(p \in P, q \in P)\}, p \neq q$, that are clip points w.r.t. b. A point in the stairline is called a stairline point.

Stairline points necessarily clip away more dead space than the skyline points p, q from which they are spliced, because they take coordinates from both p and q such that they are farthest from the corner R^b.

IV. CBB-based R-trees

Section III introduced the concept of clipped bounding boxes (CBBs) and two approaches to defining clip points for them. Here we describe how to integrate them into arbitrary R-tree variants. Recall that the R-tree variants vary in how they determine the contents of their nodes, but not in their general layout; thus, our extensions here apply to any variant.

A. Layout and structure of clipped R-trees

Figure 3 extends the example of Figure 2, contrasting a classic MBB-based R-tree (a) with a clipped one (b). In Figure 3a, the seven spatial objects (o_1–o_7) are indexed using the traditional R-tree with $M = 5$ and $m = 2$, which results in a two-level hierarchy of MBBs. The root node (in black) corresponds to R_0, which minimally bounds the two leaf nodes (in green). The R-tree distributes the objects into nodes well, as the leaves have zero overlap. The range query, Q, intersects two spatial objects: it fully covers o_6 and it partially intersects o_7. Nevertheless, because Q intersects all MBBs in the R-tree, all three nodes must be scanned, which unfortunately includes the node of R_1, inside which Q only overlaps dead space.

The clipped R-tree, for contrast, is shown in Figure 3b. Each of the MBBs (with corner points depicted by solid circles) from Figure 3a is independently augmented with clip points (depicted by hollow circles). Six clip points are introduced: two in R_0 (the black ones), three in R_1, and one in R_2. While Q still intersects R_0 and R_2 as they contain objects that are within Q's range, the excess leaf-node scan of R_1 is averted.

Figure 4a illustrates a typical R-tree data structure. The directory nodes (in this case just the root) contain an array with an MBB (R_0) followed by a list of between m and M children. For each child, its MBB and a pointer to the child

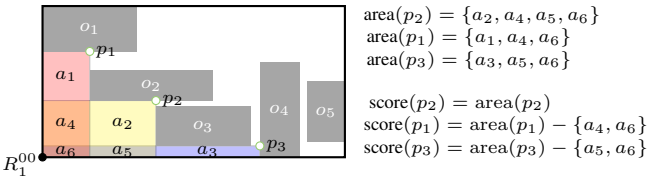


Fig. 5: Demonstration of the overlap approximation. The combined score of $\{p_1, p_2, p_3\}$ overcounts the overlap of $\{p_1, p_2, p_3\}$ and undercounts the overlap of $\{p_1, p_3\}$, but these often correspond to the same area, a_6.

node are given. The leaves contain an array with an MBB and a list of pointers to actual objects. To clip the R-tree, *we retain this original structure exactly* and augment it with the structure in Figure 4b that contains the clip points. A directory table is indexed by the ids of the R-tree nodes: entry 1 corresponds to R_1. It contains a length, since we adaptively determine the variable number of clip points per node, and a pointer to an array. Inside the array, the bitmask of the clip point is given first and the actual coordinates of the point are given next. The clip points are ordered by the volume that they clip away in order to detect non-intersection as quickly as possible.

B. Constructing clipped bounding boxes

In Section III, we described two sorts of clip points, giving rise to two different clipped bounding boxes. Here, we describe how *to compute* those CBBs. In particular, we focus on the challenging problem of selecting k clip points from a set of choices that grows with M. We discuss this k-selection problem first, then turn to the high-level clipping algorithm.

Selecting k clip points Determining the optimal selection of k clip points would incur exponential cost, because there is no known algorithm for $\geq 3d$ better than enumerating all possible size-k subsets [32]. However, we can avoid this cost with three reasonable simplifying assumptions: (1) each corner of a CBB is independent; (2) the point that independently clips away the most volume is among the k choices; and (3) if multiple points clip away area in the same corner, the overlap of that area is small and/or covered by the point in assumption (2).

We make assumption (1) because if clip points p and q arise from different corners, they can only clip away area that overlaps each other if there is substantial dead space. In that case, optimality is not crucial: greedily selecting k clip points will prune a lot of dead area, anyway. The simplification permits linearly combining the solutions from the 2^d corners, rather than computing cross-corner overlap.

Figure 5 illustrates the intuition behind assumptions (2) and (3). Three potential clip points, p_1–p_3, for corner R^{00} are shown, along with the area that each would clip away, $\text{area}(p_i)$. If we pick multiple clip points for this corner, assumption (2), accurately in this case, asserts that p_2 would be in the optimal solution. Assumption (3) asserts that the overlap in clipped area among the multiple points will be contained in $\{a_2, a_4, a_5, a_6\}$ or very small. More specifically we assume that the point clipping away the largest volume (in this case p_2) will be selected *and* will clip away the overlap for all chosen points. Thus we assume that for another p_i, it will contribute $\text{area}(p_i) - \text{area}(p_i \cap p_2)$ to the overall union. Whereas exact computation would require invoking the exponential-cost inclusion-exclusion principle, we simply add these scores together to approximate the clipped area.

In this example, our approach produces the exact score. This occurs because p_1 and p_3 lie on opposite sides of p_2, which commonly happens: the point clipping away the most area is likely to lie on the diagonal and the next best choices (for small k) are likely to be on either side. Even when this intuition fails, the error of approximation is bounded by the intersection of the smaller rectangles, which itself is small.

As a last optimisation, we elect to only store clip points that clip away an additional $\geq \tau$ % of the volume; otherwise, they increase our storage cost without having a high likelihood of containing a query rectangle. Thus, we could end up with fewer than the intended number of k clip points if they do not prune much dead space, anyway.

Algorithm 1 Clip: (node N, k, τ) $\longrightarrow$ set of clip points C

```
1:  L ← empty list of clip points
2:  for each bitmask b ∈ 0 ... (2^d − 1) do
3:      P ← S_b({o_i^b : o_i ∈ N.children})
4:      if using stairline points then
5:          for each s_i, s_j ∈ P do
6:              if ∀s_k ∈ P, ✪_{∼b}(s_i, s_j) ⊀_b s_k then
7:                  P' ← P' ⋃ {✪_{∼b}(s_i, s_j)}
8:          P ← P'
9:      assign scores ∀p ∈ P as in Figure 5
10:     for each p ∈ P, with p.score > τ × area(N) do
11:         L.append(⟨p, b⟩)
12: return the min(k, |L|) clip points in L with highest score
```

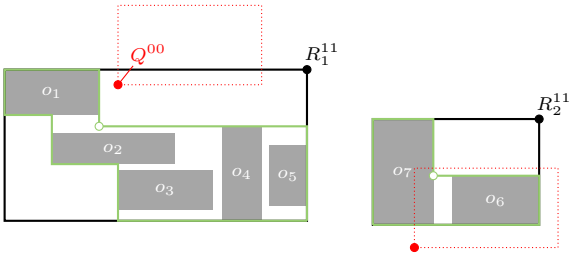
The clipping algorithm To construct a clipped R-tree, we apply Algorithm 1 to every tree node and store each result with an entry in the auxiliary structure (Figure 4b). We iterate over the corners of the MBB independently (Line 2 and assumption 1), computing for each the skyline of the object corners (Line 3). We optionally splice the skyline points (Lines 4–8) to clip away more area. Then, we determine which clip point clips away the most area and assign approximate scores to the rest (Line 9 and assumptions 2 and 3). We finish processing the corner by keeping only the clip points passing the τ threshold (Lines 10–11). After iterating every corner, we sort the clip points by score and return the k highest-scoring.

C. Querying a clipped bounding box

Since CBBs use the corners of their MBBs, intersection can be done very efficiently, as shown in Algorithm 2. Lines 3–5 transform a typical MBB intersection test into a clipping-enabled one. Given clip points C, we simply check dominance (Definition 4) between the corner of the query rectangle $Q^b \in Q$ obtained by $Q^{\sim \text{c.mask}}$ and each clip point $c \in C$. (For queries, *selector* is fixed to $2^d - 1$: the *xor* expression ($\oplus$) is equivalent to negating *c.mask*. Its purpose will be clarified in

Algorithm 2 Intersection Test: $(R, C, Q, \text{selector}) \longrightarrow \texttt{bool}$

```
1: if Q ∩ R = ∅ then
2:     return FALSE
3: for each c ∈ C do
4:     if Q^(selector⊕c.mask) ≺_c.mask c.coord then
5:         return FALSE
6: return TRUE
```



(a) Querying the bottom node. (b) Querying the top node.

Fig. 6: Using *dominance* to test CBB intersections. The hollow point gives the bitmask of the clip point (c.mask); the solid red point shows the corner of Q ($Q^{\sim\text{c.mask}}$) with which to check dominance with respect to the solid, black point ($R^{\text{c.mask}}$).

Section IV-D.) Intuitively, it is the least "competitive" query corner that can dominate a given clip point. If any clip point is dominated by Q, then the CBB and Q are disjoint.

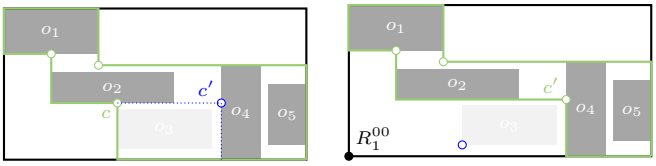
We query each leaf node of our running example in Figure 6. In Figure 6a, we compare Q to the first clip point, which is paired to corner R_1^{11}. (The clip points, recall, are sorted by the volume-based score.) Because Q^{00} dominates the clip point, relative to R_1^{11}, we know that the part of Q inside the MBB lays inside dead space and avoid scanning R_1. On the other hand, in Figure 6b, Q^{00} *does not* dominate the (sole) clip point; so, we can conclude that Q intersects the CBB of R_2.

D. Updating a clipped bounding box

An update to a memory-resident CBB is cheap, as a disk write to persist the changes in the underlying data is always coincident. Still, many unnecessary updates can be avoided.

First, we observe that any update that affects x MBBs can affect at most $x + 1$ CBBs. Unlike with the boundaries of MBBs, the changes to clip points do not need to propagate up the tree, because each MBB is clipped independently. If the last MBB change occurs at level l of the tree, then the next shallower level, $l+1$, is the last level at which clip points may change, as the clip points are based on the MBBs of the level below. Moreover, a split/merge that does not change an MBB also does not change the corresponding CBBs.

In general, if the MBB of node n changes, we recompute the clip points of n, because our thresholding with τ and our top-k ordering and selection are both distorted by the change: one must re-examine all candidate clip points for at least one corner, anyway. Below, we discuss how to avoid the additional $x+1$'st CBB change for deletions and insertions. Modifications are handled by deleting and then re-inserting the object.



(a) Deleting object o_3. (b) Inserting object o_3.

Fig. 7: Clip points before (green) and after updates (blue). Deleting o_3 creates a better clip point c', but c is still valid. On insertion, the blue corner of o_3 dominates c' with respect to the solid, black point, indicating that o_3 invalidates c'.

Deletions Deletions are the easier case, because they *generate new dead space*; thus, we can handle them "lazily". Figure 7a illustrates the deletion of object o_3 from our running example, which affects the CBB, but not the MBB, of the bottom node. Prior to the deletion, clip point c is in C. After the deletion, we could replace c with c', which prunes the new dead space. However, if we continue to use clip point c instead of c', we obtain accurate query results with the same pruning rate as prior to the deletion. Moreover, a subsequent insertion (as shown next), whether to a new object or as part of modifying o_3, only needs to be handled if it intersects the dead space pruned by the previous point c. Therefore, on deletions we never need to change a CBB if the MBB is unaffected.

Insertions Insertions *remove dead space*, potentially invalidating our clip points; so, we handle them "eagerly". For each insertion, we check whether it invalidates one of our top-k clip points. If so, we recompute the CBB. If not, we are certain that our top-k selection is still correct, because the score of every other candidate clip point is either lowered or unaffected by the insertion. Figure 7b illustrates the insertion of o_3 (had it not been there before). Observe that c' prunes away space occupied by o_3, yielding incorrect query results. This can be detected by testing whether the blue corner 00 of o_3 dominates c' with respect to the black corner R_1^{00} of the MBB.

The validity test is identical to our query in Algorithm 2, except that it selects a different corner of the "query" rectangle, o_3, by fixing *selector* to 0 (i.e., selecting the *same* corner as c.mask). Whereas a query checks whether an entire rectangle is contained in the dead space pruned by a single clip point, an insertion checks whether any part of a rectangle is contained in that dead space. If the intersection test returns FALSE, then the newly inserted object dominates the clip point (i.e., part of the object is prunned away by the clip point) and thus is invalid. (Inserts propagate up from the leaves, so always $Q \cap R \neq \emptyset$.) If the intersection test returns TRUE, the top-k clip points are unchanged and the CBB does not need to be updated.

V. Experimental Evaluation

We incorporate our clipping algorithms from Section IV into a variety of R-tree variants of the existing benchmark for multi-dimensional indexes [33]. The benchmark has been used in many spatial indexing studies (e.g., [13]). We then evaluate query/update performance, overhead, and spatial join performance relative to unclipped (i.e., unmodified) R-trees.

A. Environment and experimental setup

Spatial Indexes We use `C` implementations of four popular R-tree variants [13], including the quadratic R-tree (QR-tree) [5], Hilbert R-tree (HR-tree) [23], R*-tree (R*-tree) [12], and revised R*-tree (RR*-tree) [13]. We modify each implementation to construct clip points (as per Algorithm 1) for each node prior to flushing the node to disk and to utilise the clip points for intersection tests (as per Algorithm 2). We configured each index using the values for min and max node capacities (M and m) as described in [13]. We observe minimal effect from varying k and τ; they are set to values of $k = 2^{d+1}$ and $\tau = 2.5\%$. (Figure 10 illustrates the effect of k nonetheless, but we lack space to also vary τ.) To differentiate the two CBBs, we denote the skyline-only approach from Section III-B as C^{SKY} and the point-splicing approach from Section III-C as C^{STA}.

Hardware All experiments use a commodity desktop with a quad-core Intel Core i7-3770 3.4 GHz CPU, 16GB of physical memory, and a 500GB 7200RPM hard disk. The desktop runs Ubuntu 16.04 LTS (kernel version 4.4.0) and the code is compiled with `gcc` (version 5.4.0).

B. Datasets and queries

Datasets For the main evaluation, we use seven spatial datasets, ranging from 2–3 dimensions. Four challenging (two real and two synthetic) datasets are taken from the existing benchmark [33]. The real datasets include: `rea02` (a $2d$ dataset of 1 888 012 rectangles and points representing street segments in California) and `rea03` (a $3d$ dataset of 11 958 999 points representing three floating point attributes in a biological data file). The synthetic datasets are `par02` and `par03` containing 1 048 576 $2d$ and $3d$ boxes, respectively. The objects are generated with a very large variance in size and shape, which makes them challenging to approximate.

In addition to the above datasets, we add three new datasets stemming from our main use case and collaboration with neuroscientists in the Human Brain Project [34]. The datasets contain volumetric boxes representing different spatial objects in a $3d$ brain model: 2 570 016 segments of axons (`axo03`), 1 288 251 dendrites (`den03`), and 3 858 267 neurites (`neu03`).

Queries We query data as in [33]. Given dataset D and number of result objects $|R|$ as input, the generator produces queries originating from the dithered centers of the objects in D. $|R|$ object centers are chosen randomly so that the most dense data regions are also most actively queried. For each dataset, we produce three query profiles of varying selectivity: $QR0$, $QR1$, and $QR2$ retrieve approximately 1, 10 and 100 objects, respectively, per query.

C. Results and discussion

Bounding object comparison We begin by evaluating how well the clip points (CBB^{SKY} and CBB^{STA}) eliminate dead space, relative to six alternative bounding objects (studied earlier in [6], [20]). Figure 8 illustrates each shape on our running example. We compute minimum bounding circles (MBC) as per Welzl [30] and rotated minimum bounding boxes (RMBB) by iterating the edges of the convex hull (CH) and computing the minimum bounding box with the same orientation as each edge. The m-corner polygons (4-C and 5-C) are the smallest-area polygons with $\leq m$ corners that fully bound the children, computed similarly to [35]. Finally, the convex hull (CH) is computed using Graham Scan [36]. Observe the extremes that apply in general: CH lower bounds the dead space for all the convex polygons but has the highest representation overhead ($\mathcal{O}(n)$ corners). MBC and MBB have the lowest representation overheads ($\leq$ two points) but coarsely approximate their contents.

	QR0	QR1	QR2	Total
QR-tree	24/44	16/29	7/13	16/29
HR-tree	25/42	18/30	8/14	17/29
R*-tree	21/38	15/28	7/14	14/27
RR*-tree	15/28	11/21	4.5/9.5	10/19
Total	21/38	15/27	6.5/13	14/**26**

TABLE I: Average improvement in % I/O reduction using skyline/stairline clipping for each R-tree.

Figure 9 contrasts the eight bounding objects in terms of area (left) and representation overhead (right). Following [6], [20], we restrict to $2d$ datasets (just this figure), as we know of no way to calculate minimum bounding m-corner polytopes in higher dimensions. The simplicity with which MBC, MBB, and CBB generalise to higher dimensions is a clear advantage.

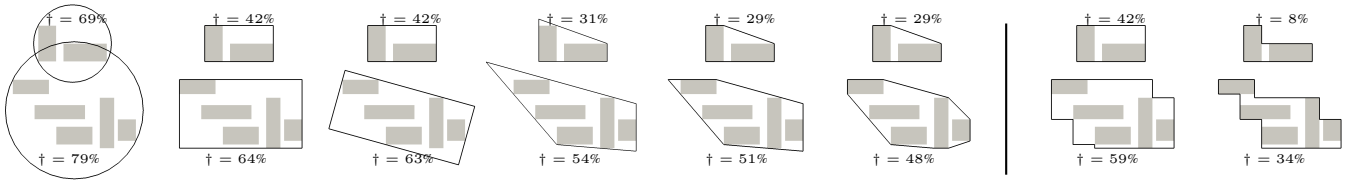
For each dataset, we built an RR*-tree.[1] For each node of the RR*-tree, we replace the MBB with a new minimum bounding shape and measure its area. Figure 9(a) reports the percentage of this area that is empty, averaged over all nodes, for each bounding shape. The number of points used is reported in Figure 9(b). As expected, the convex polygons prune more dead space as the number of corner points increases. CBB^{SKY} is generally competitive with 4-C using only one or two clip points on average. (The reported cost in Figure 9(b) includes the two corner points of the original MBB.) CBB^{STA}, using up to 3.4 clip points on average, outperforms even CH, which uses on average 12.5 and 11.8 points.

Remaining dead space Figure 10 expands the previous experiment by evaluating the coverage of CBB in all four R-tree variants and more datasets (with some still omitted for space[2]). The total height of each bar, relative to the y-axis indicates the percentage of the MBB that is dead space; the height of the top (clear) part of the bar indicates the fraction of the dead space that is clipped away by the skyline (top) or stairline (bottom) points. If the solid, lower part of the bar is comparably short, then most dead space is eliminated.

Along the x-axis, we vary three parameters: the highest granularity groups correspond to a given dataset; the four groups within that are colour- and pattern-coded by R-tree variant; at the smallest granularity, we vary k, the maximum

[1]The RR*-tree gives the most pessimistic results for CBB of all variants.

[2]The point-only `rea03` dataset essentially occupies zero volume; so, the entire MBB is dead space at the leaf node level and the experiment is not very informative. The results for `den03` and `neu03` are very similar to `axo03`.



(a) MBC (b) MBB (c) RMBB (d) 4-C (e) 5-C (f) CH (g) CBB$^{\text{SKY}}$ (h) CBB$^{\text{STA}}$

Fig. 8: Visualization of different bounding methods over the two leaf nodes from Figure 3a and their dead space (†).

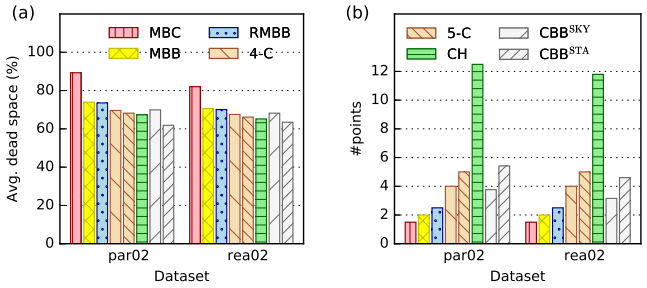


Fig. 9: Comparing different bounding methods w.r.t dead space (left) and storage requirements (right).

number of clip points stored per node. We vary k from 1 to 2^{d+1} (i.e., up to twice the number of corners), although reiterate that this is a maximum bound per node: clip points with a score less than $\tau = 2.5$ % are not indexed.

Analysis We first consider broad trends across datasets. All R-tree variants produce bounding boxes on the neuroscience dataset (`axo03`, far right) that are mostly dead space. In fact, for all four datasets, nodes on average contain ≥ 60 % dead space. The increase from the $2d$ synthetic `par02` dataset to the similarly generated $3d$ dataset illustrates the known fact that bounding boxes become poorer approximations of their contents as the dimensionality increases. Despite different packing algorithms, the QR-tree, R*-tree, and RR*-tree produce similar occupancy rates in their bounding boxes. This is generally less than the HR-tree (the exception being the neuroscience dataset where all four have extremely high dead space rates). We observe that most of the dead space is caused by (the much more frequent) leaf-level nodes (not shown independently), which must bound the actual spatial objects using a rough box; at higher levels of the tree, the MBBs more tightly bound other MBBs. Overall, such voluminous dead space across R-tree variants and datasets strongly motivates this research.

Turning to the ratio of clipped dead space to remaining dead space (i.e., fraction of each bar that is filled in), we see more variation across datasets, but consistency across R-tree variants. It is more difficult to clip away dead space from the street segments (`rea02`), which is quite intuitive: we expect street segments to "wrap around" some of the dead space, particularly in cities with grid patterns. Nonetheless, even on this least promising dataset, we clip away more than a fifth of the dead space. For the $3d$ datasets, we clip away more than 60%, irrespective of the packing method.

Perhaps most encouraging is the fraction of dead space pruned by the first (ordered) clip point (i.e., at $k = 1$). With just one point, 26%, 23%, 22%, and 22% of dead space is clipped away by QR-tree, HR-tree, R*-tree, and RR*-tree, respectively (on `rea02`). Since we order clip points by the (heuristic) volume of dead space that they clip away, the effect of each subsequently added clip point diminishes and eventually flattens out. Nevertheless, we observe that with $k = 2^d$ the CBBs still eliminate substantial portions of dead space. This suggests that k can certainly be large enough to produce one clip per corner. Overall, clipping with up to two points per corner (i.e., $k = 8$ in $2d$ and $k = 16$ in $3d$), eliminates almost half of all dead space: 58%, 60%, 49%, and 48% is clipped away in QR-tree, HR-tree, R*-tree, and RR*-tree, respectively. Since the number of actually stored clip points is often lower (recall $\tau = 2.5\%$), we set $k = 2^{d+1}$ to this maximum bound in the following experiments.

To compare skyline- and stairline-based clipping, observe that the difference lies in the fraction of the bars that are filled in; the total height (i.e., amount of dead space) is dependent only on the dataset and packing algorithm. It is clear, as asserted in Section III-C, that the stairline points clip away much more dead space, eliminating on average almost 50% more for the same values of k.

Range Query Performance While the first experiments show that we clip away a lot of dead space, the real objective is to improve query performance. Removing dead space is only useful if that is where the query rectangle intersects the MBB. Here we evaluate how well clipping reduces I/Os. Following numerous studies on disk-based indexing (e.g., [13]), we assume that internal (non-leaf) nodes are memory-resident and measure the number of leaf-level nodes accessed as our default I/O metric. (Later we remove this assumption.)

Figure 11 reports I/Os for stairline-based clipped R-trees. Query selectivity decreases in the subfigures from left to right. Within each subfigure, four vertical bars corresponding to each R-tree variant are grouped by dataset. The bar height reports the percentage of I/Os relative to not clipping.

Analysis We focus on stairline clipping, but similar (dampened) trends apply to the skyline points. For example, skyline/stairline points clip away 20%/39% of RR*-tree volume, reducing I/Os by 10%/19%, i.e., C$^{\text{STA}}$ performs $\approx 2\times$ better.

On the most selective queries (a), we consistently reduce I/Os by $\geq$ 10%, even exceeding 50% in some cases. The gains are particularly pronounced on the neuroscience datasets (`axo03`, `den03`, and `neu03`), where we save 40–50 % of I/Os. While the difference in the amount of dead space clipped away in Figure 10 was relatively small between the

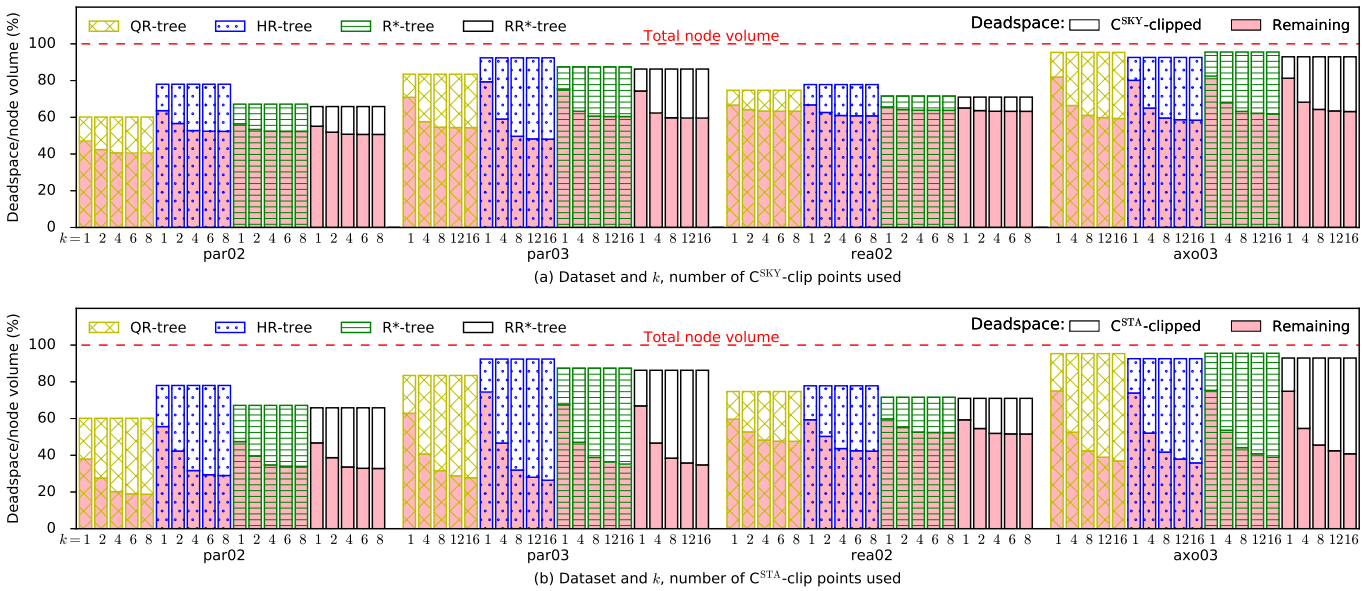


Fig. 10: Average dead space per node and R-tree for skyline- (above) and stairline-based (below) clipping.

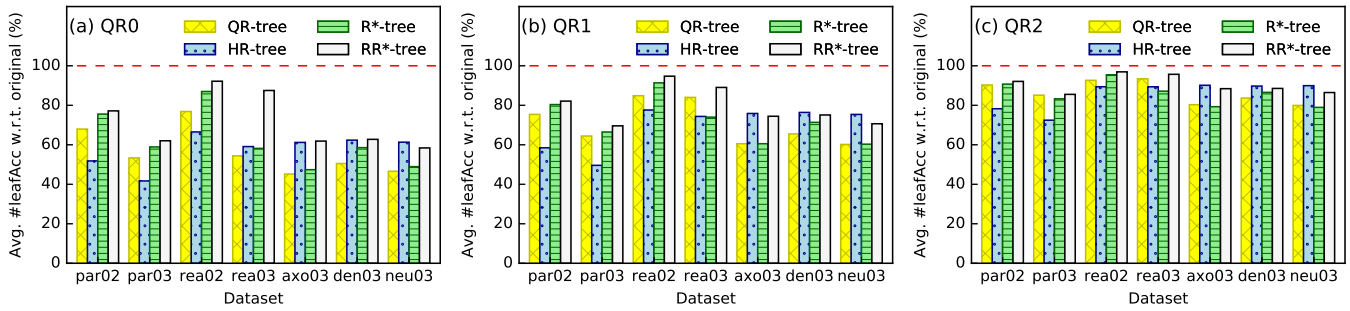


Fig. 11: Average #leaf accesses in clipped R-trees w.r.t. their unclipped counterpart (100%) for stairline-based clipping.

HR-tree and the other R-tree variants, here we observe a much larger difference. The RR*-tree gains somewhat less query performance from clipping than the other variants (19% versus 27–29% as an average over all query profiles). We attribute that to its already strong query performance (shown in [13]).

The observed performance gains diminish with decreasing query selectivity. This is expected, as the fraction of spatial objects that are around the query boundary decreases with larger query ranges (i.e., lower selectivity). Nevertheless, HR-tree and QR-tree still benefit appreciably (circa 20%) on the `rea02`/`rea03` and neuroscience datasets, respectively. The RR*-tree gains relatively less than the other variants as query selectivity decreases, and on QR2, the difference between variants is not very pronounced.

Table I averages the performance gains of Figure 11 across all datasets. For QR0, average relative I/Os drops to 56%, 58%, 62%, and 72% for QR-tree, HR-tree, R*-tree, and RR*-tree, respectively. In total, clipping MBBs results in an average reduction of 26% in I/Os, considering all datasets, query profiles, and R-tree variants.

Update cost Next, we quantify how effective are the Section IV-D strategies for avoiding unnecessary re-clipping of CBBs. Recall that we never re-clip on a deletion if the MBB is not changed, so this experiment focuses on insertions.

We first randomly choose 90 % of the input file to batch-construct the clipped R-tree variants. Then, we execute our insertion routine for each of the remaining 10 % of objects. We report on the y-axis of Figure 12 the *expected number of re-clips per insertion*: i.e., the number of nodes that we re-clip divided by $0.1\times$ the input file size. Along the x-axis, we vary dataset and R-tree variant. Each stacked bar shows the cause of the re-clip: at the bottom are node splits, which always force an MBB recomputation. In the middle are MBB changes without a node split, which force a CBB change. The top part shows CBB changes with no corresponding change to the MBB, i.e., when Algorithm 2 returns `FALSE`.

Analysis Without the Section IV-D strategy for avoiding unnecessary re-clips, the number of CBB changes per insert would be exactly 1.0 higher than the number of MBB changes. However, we observe far fewer re-clips than this worst case. Averaging across all datasets, ≤ 0.35 of a node needs to be re-clipped per insert, with the exception of the R*-tree (discussed below). Generally, ≈½ the re-clips are caused by underlying MBB changes. The challenging neuroscience datasets give the highest re-clip rates, where we still avoid ≈ 60 %.

With respect to d, the $2d$ datasets have lower re-clipping

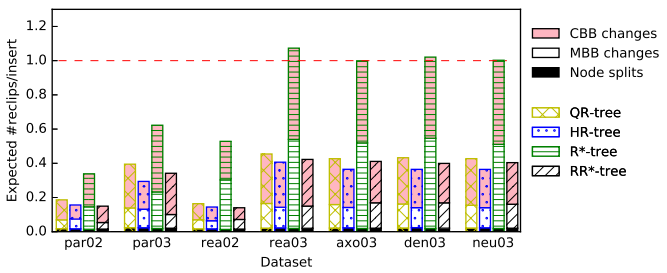


Fig. 12: Expected number of re-clipped CBBs per insertion.

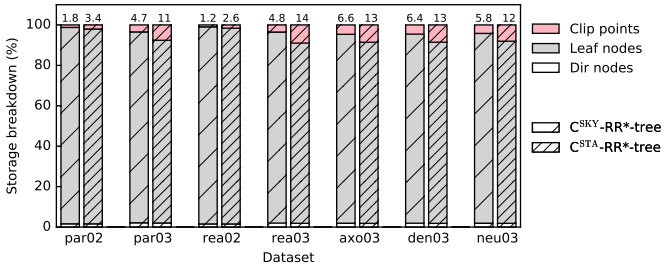


Fig. 13: CBB storage overhead.

rates than their $3d$ counterparts (observe `par0*` and `rea0*`). The neuroscience datasets, with long, skinny objects, have similar re-clipping rates to `rea03`. Considering the R*-tree, it consistently suffers the most re-clips, owing to its reinsertion policy: a node split re-inserts every entry of that node; thus, single-object inserts often cause many MBB/CBB changes.

Extra storage cost Figure 13 reports the increased storage requirements for clipped RR*-trees ($k = 2^{d+1}, \tau = 2.5\%$). Each bar decomposes the percentage of bytes devoted to directory nodes, leaf nodes, and clip points. For each dataset (x-axis), we show results for C^{SKY} (left) and C^{STA} (right). As we only retain clip points with scores $\geq \tau$, we report atop each bar the average number of clip points that are stored.

Analysis Overall, the CBB overhead is quite low and storage is dominated by the far more frequent leaf nodes. The storage dedicated to clip points never exceeds 2% ($2d$ datasets) nor 9% ($3d$ datasets), irrespective of which method (skyline or stairline) is used to generate them. This confirms that clip points and internal nodes can generally be memory-resident, as they contribute just a few percent of the total storage.

No dataset averaged all $k = 2^{d+1}$ clip points. As few as 6 (C^{SKY}) and 13 (C^{STA}) clip points are stored per node in the $3d$ neuroscience datasets; ≤ 3 clip points are averaged on the $2d$ datasets. This reflects that some objects are often near to MBB corners; dead space is not uniformly distributed.

C^{SKY} produces fewer clip points than C^{STA} as its clip points prune less area, often $< \tau$. Thus, C^{SKY} has a smaller overhead than C^{STA} (3-fold on `rea03`), but worse I/O performance.

Clipping CPU cost During construction, we recommend an R-tree node is clipped in main memory just prior to flushing it to disk so that the main memory cost is subsumed by that of the disk write. Nevertheless, for completeness, we quantify the main memory overhead for clipping. To do so, we configure

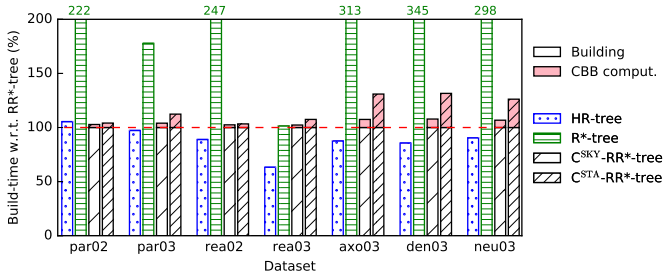


Fig. 14: Index building and CBB computation overhead.

buffer sizes for all R-trees so that they are completely memory-resident and then measure the resultant actual CPU time.

Figure 14 reports the CPU construction time relative to an unclipped RR*-tree (100%). Datasets vary along the x-axis. HR-tree is generally the fastest to build (due to its bulk-loading) and R*-tree is generally the slowest to build (due to its forced reinsertion of items during node split). These two unclipped variants provide context for interpreting the clipping overhead. The shaded part (above the dashed line) of the stacked bars for the clipped RR*-trees shows the component of the construction time devoted to clipping. When exceeding 200%, the value is written above the bar.

Analysis As k grows exponentially with d, so too does the overhead of CBB computation. For the spatial datasets (i.e., $d \leq 3$), C^{SKY} adds $< 7\%$ extra computation time. C^{STA} clipping is more expensive, adding up to 4% and 30% of extra computation in $2d$ and $3d$ datasets.

Spatial Join Performance We perform a join using `axo03` and `den03`, resulting in 1 985 969 pairs, using stairline points and two join strategies: Index Nested Loop Join (INLJ) is applied when only one dataset is indexed and Synchronised Tree Traversal (STT) is applied when both datasets are indexed [8].

Analysis In the INLJ evaluation, we build an index on the larger dataset (`axo03`) and probe it with every object from `den03` (essentially one range query per `den03` object). The results mirror those of the range query experiments: clipping reduces I/Os by 40%, 53%, 50%, and 39% in the HR-tree, QR-tree, R*-tree, and RR*-tree, respectively.

For the STT strategy, we recursively restrict the search space to the intersection of the CBBs of the corresponding sub-trees and apply dominance tests to check whether a child CBB falls within it. We measure leaf accesses for both trees and observe a 17%, 20%, 20%, and 16% reduction in I/Os in the HR-tree, QR-tree, R*-tree, and RR*-tree, respectively. Note that the obtained reduction is lower than in the case of INLJ, but STT performs significantly better than INLJ, having a lower total number of accesses. E.g., in the case of the RR*-tree, INLJ with clipping incurs around 4×10^6 leaf accesses whereas STT with clipping only around 10^6. Intuitively, the intersect area of two tree nodes in STT is larger than the intersect area of a tree node with a single data object in INLJ and thus there is a lower chance that it will be fully enclosed within dead space.

Scalability Experiment Our last experiment scales up the synthetic data to 2^{30} objects so that it exceeds our machine's

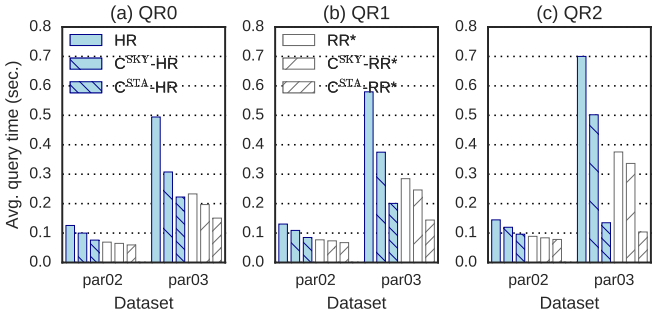


Fig. 15: Querying 1 billion object datasets.

16 GB of physical memory (yielding 71 and 96 GB for the `par02` and `par03` RR*-tree index disk dumps, respectively, with similar sizes for HR-tree). Starting with all indexed data on disk and nothing buffered we measure the query time for 500 random queries on each query profile, allowing the OS to cache paths for previously touched nodes. The average query run times are reported in Figure 15 for each selectivity.

Analysis Both skyline and stairline CBBs boost query performance in the HR-tree and R*-tree. Matching I/O performance in Figure 11, C^{STA} clipping averages $2\times$ as efficient as C^{SKY}. Interestingly, a C^{STA}-clipped HR-tree matches (in QR0 and QR1) or even outperforms (in QR2) an unclipped RR*-tree. In all, spatial search with CBBs, even for 1 billion objects, reaches interactive times—200 ms or less.

VI. Conclusion

Minimum bounding boxes (MBBs) are ubiquitously used in spatial indexing to represent a set of spatial objects. However, they often enclose significant "dead space" that contains no actual objects. This paper proposed *clipping away* empty corners of MBBs with a lightweight overhead. Each auxiliary "clip point" defines a large, empty rectangular area that can be discarded with a single point comparison. The resultant bounding shapes are simple but non-convex, thereby pruning more area than previously proposed alternatives to MBBs.

We plugged our skyline- and stairline-based clipping strategies into four R-tree variants of a well-known experimental benchmark. Compared to unclipped R-trees, the $(2\times)$ more aggressive stairline points removed $\geq 27\%$ of the dead space. Irrespective of R-tree variant, this translated into $\approx 26\%$ I/O reduction on average across all workloads. With a storage overhead of a few percent, clipped bounding boxes are highly effective for accelerating spatial data processing.

Acknowledgment

We thank Dr. Satya Valluri, a former DIAS lab member, for insightful feedback during the early stages of this work. This project received funding from the European Union's Horizon 2020 research and innovation programme through projects *HBP SGA1*, GA No 720270, (EPFL) and *CoupledDB*, Ga No 753810, (NTNU) and from the Norwegian Research Council *ExiBiDa* project.

References

[1] R. K. V. Kothuri, S. Ravada, and D. Abugov, "Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data," in *SIGMOD*, 2002, pp. 546–557.
[2] IBM Informix Dynamic Server v12.1 Information Center, *IBM Informix R-tree Index, User's Guide*, March 2013.
[3] *PostGIS 2.0 Manual*, http://postgis.net/docs/manual-2.0.
[4] V. Pandey, A. Kipf, D. Vorona, T. Mühlbauer, T. Neumann, and A. Kemper, "High-performance geospatial analytics in hyperspace," in *SIGMOD*, 2016, pp. 2145–2148.
[5] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.
[6] T. Brinkhoff, H.-P. Kriegel, and R. Schneider, "Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems," in *ICDE*, 1993, pp. 40–49.
[7] H. Samet, "The quadtree and related hierarchical data structures," *ACM Comput Surv*, vol. 16, no. 2, pp. 187–260, 1984.
[8] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using R-trees," in *SIGMOD*, 1993, pp. 237–246.
[9] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," in *SIGMOD*, 1996, pp. 259–270.
[10] S. Brakatsoulas, D. Pfoser, and Y. Theodoridis, "Revisiting R-tree construction principles," in *ADBIS*, 2002, pp. 149–162.
[11] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A dynamic index for multi-dimensional objects," in *VLDB*, 1987, pp. 507–518.
[12] 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*, 1990, pp. 322–331.
[13] N. Beckmann and B. Seeger, "A revised R*-tree in comparison with related index structures," in *SIGMOD*, 2009, pp. 799–812.
[14] H. V. Jagadish, "Spatial search with polyhedra," in *ICDE*, 1990, pp. 311–319.
[15] O. Gunther, "The design of the cell tree: An object-oriented index structure for geometric databases," in *ICDE*, 1989, pp. 598–605.
[16] N. Katayama and S. Satoh, "The SR-tree: An index structure for high-dimensional nearest neighbor queries," in *SIGMOD*, 1997, pp. 369–380.
[17] D. A. White and R. Jain, "Similarity indexing with the SS-tree," in *ICDE*, 1996, pp. 516–523.
[18] P. van Oösterom and E. Claassen, "Orientation insensitive indexing methods for geometric objects," in *SDH*, 1990, pp. 1016–1029.
[19] O. Danko and T. Skopal, "Elliptic indexing of multidimensional databases," in *ADC*, 2009, pp. 85–94.
[20] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger, "Multi-step processing of spatial joins," in *SIGMOD*, 1994, pp. 197–208.
[21] S. Börzsönyi, D. Kossman, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.
[22] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, *R-Trees: Theory and Applications*. Springer, 2005.
[23] I. Kamel and C. Faloutsos, "Hilbert R-tree: An improved R-tree using fractals," in *VLDB*, 1994, pp. 500–509.
[24] S. T. Leutenegger, M. Lopez, and J. Edgington, "STR: A simple and efficient algorithm for R-tree packing," in *ICDE*, 1997, pp. 497–506.
[25] 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 *SIGMOD*, 2004, pp. 347–358.
[26] V. Gaede and O. Günther, "Multidimensional access methods," *Comput Surv*, vol. 30, no. 2, pp. 170–231, 1998.
[27] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami, "Geometric computing and the uniform grid data technique," *Comput Aided Design*, vol. 21, no. 7, pp. 410–420, 1989.
[28] C. L. Jackins and S. L. Tanimoto, "Oct-trees and their use in representing three-dimensional objects," *Comput Vision Graph*, vol. 14, no. 3, pp. 249–270, 1980.
[29] D. B. Lomet and B. Salzberg, "The hB-tree: A multiattribute indexing method with good guaranteed performance," *ACM Trans Database Syst*, vol. 15, no. 4, pp. 625–658, 1990.
[30] E. Welzl, "Smallest enclosing disks (balls and ellipsoids)," in *New Results and New Trends in Computer Science*, 1991, pp. 359–370.
[31] S. Wang, D. Maier, and B. C. Ooi, "Fast and adaptive indexing of multi-dimensional observational data," *PVLDB*, vol. 9, no. 14, pp. 1683–1694, 2016.
[32] K. Bringmann, T. Friedrich, and P. Klitzke, "Two-dimensional subset selection for hypervolume and epsilon-indicator," in *Proc. GECCO*, 2014, pp. 589–596.
[33] N. Beckmann and B. Seeger, *A Benchmark for Multidimensional Index Structures*, http://www.mathematik.uni-marburg.de/~seeger/rrstar/.
[34] H. Markram *et al.*, "Introducing the Human Brain Project," *Procedia Computer Science*, vol. 7, pp. 39–42, 2011.
[35] A. Aggarwal, J. S. Chang, and K. Y. Chee, "Minimum area circumscribing polygons," *Visual Comput*, vol. 1, no. 2, pp. 112–117, 1985.
[36] R. L. Graham, "An efficient algorithm for finding the convex hull of a finite planar set," *Inform Process Lett*, vol. 1, no. 4, pp. 132–133, 1972.