Efficient Spatial Joins for Large Sets of Geometric Objects

Hannah Bast
University of Freiburg
Freiburg, Germany
bast@cs.uni-freiburg.de

Patrick Brosi
University of Freiburg
Freiburg, Germany
brosi@cs.uni-freiburg.de

Johannes Kalmbach
University of Freiburg
Freiburg, Germany
kalmbach@cs.uni-freiburg.de

Axel Lehmann
University of Freiburg
Freiburg, Germany
lehmann@cs.uni-freiburg.de

ABSTRACT

We consider the following problem: Given two sets of geometric objects in 2D (points, lines, polygonal areas, and collections of these), compute the set of object pairs for each of the spatial relations *contains*, *covers*, *within*, *intersects*, *touches*, *crosses*, and *equals*. We provide an efficient algorithm together with a fully functional implementation that is practical also for very large inputs. In particular, we can compute the self join of the 1.3 billion geometries from the complete OpenStreetMap data (with a total result size of 52.1 billion triples) in 1.5 hours on a standard PC. This is more than 500 times faster than the widely used PostgreSQL+PostGIS. Beyond our code, which is freely available on GitHub, we provide an extensive empirical evaluation on a variety of datasets. In particular, we investigate the effect of various speed-up heuristics and when and why PostgreSQL+PostGIS is comparatively slow.

KEYWORDS

Spatial Joins, Spatial Databases, OpenStreetMap

ACM Reference Format:
Hannah Bast, Patrick Brosi, Johannes Kalmbach, and Axel Lehmann. 2024. Efficient Spatial Joins for Large Sets of Geometric Objects. In *32nd International Conference on Advances in Geographic Information Systems (SIGSPATIAL '24), October 29–November 1, 2024, Atlanta, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/?

1 INTRODUCTION

Many queries in geospatial databases involve computing the spatial relation between geometric objects. For example, consider the following queries on the OpenStreetMap data, which as of this writing contains around 1.3 billion geospatial objects:

All streets in Germany
All strees crossing a power line
All restaurants near a transit stop

Let us quickly explain how a typical geospatial database, like the widely used PostgreSQL+PostGIS, processes such queries. As a pre-computation, a geometric index is built, typically an R-tree over all the geometric objects stored in the database. At query time, this index is used to prune combinations of objects based on their bounding boxes. For example, when the intersection of the bounding box of a street and the bounding box of Germany is empty, that pair cannot belong to any of the seven relations named in the abstract. For each of the remaining candidates, the exact geometric relations have to be computed or inferred. For example, after the described pruning via the geometric index, the first query above leaves 19.2 million streets, for each of which we have to determine whether they are fully covered by the shape of Germany.

In this paper, we are particularly interested in *self joins*. These are queries of the following kind, where *ST_Intersects* can be any of the seven spatial relations mentioned in the abstract:

```
SELECT * FROM objects A, objects B
WHERE ST_Intersects(A.geom, B.geom);
```

Such queries are interesting in three respects: (1) They are useful in themselves in that they occur as typical queries or parts thereof. (2) They tend to be the computationally hardest spatial joins, and therefore serve as a natural performance benchmark for a spatial database. (3) They can be used to precompute spatial relations, in order to speed up queries involving spatial joins.[1]

Any algorithm for self joins can also be used for a join between two non-identical sets: just compute the self-join of the union of the two sets and keep only those pairs, where one object is from the one set the other object is from the other. In our evaluation, we also consider such spatial joins. We find that PostgreSQL+PostGIS is reasonably efficient for spatial joins with relatively small candidate sets or when most objects have simple geometries, but practically unusable for self joins of large datasets with many complex geometries. Our goal is to be efficient for all such spatial joins.

1.1 Problem Definition

We consider the problem of computing the *spatial join* between two sets O_1 and O_2 of geometric objects in 2D.

Each object can be either a point, a polygonal line, an area that is delineated by one or several polygonal lines[2], or arbitrary collections of these. Specifically, we assume that each object is given as a pair of an ID (an arbitrary string, unique for that object) and a Well-Known Text (WKT) string defining the geometry. For example, the object for Freiburg Hbf could be given as: *osmnode:21769883 POINT(7.8412948 47.9977308).*

The goal is to compute all pairs (o_1, o_2), with $o_1 \in O_1$ and $o_2 \in O_2$, from one of the following seven relations: *contains*, *covers*, *within*, *intersects*, *touches*, *crosses*, and *equals*. Specifically, the output is a list of triples, where each triple consists of a pair of IDs, and a string describing their spatial relation. For example, the triple expressing that Germany contains Freiburg Hbf could be output as: *osmrel:51477 contains osmnode:21769883.* The details of the geometric relations are defined by the OGC in [9], which describes a 3×3 matrix of criteria defining geospatial relations (DE-9IM).

[1]For example, each of the first two example queries from the introduction can be computed fast if the spatial relations *intersects* and *covered* are pre-computed.
[2]In particular, areas can consist of multiple rings, with holes, and then again areas insides the holes, etc.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SIGSPATIAL '24, October 29–November 1, 2024, Atlanta, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN ?.
https://doi.org/?

The details of the input and output format are not important for this paper. We just provide them here for the sake of concreteness and because that is what our own tool uses.

1.2 Contributions

We consider the following as our main contributions:

- We provide an algorithm and a fully functional implementation for computing the spatial join between two given sets of geometric objects in 2D, according to the definition above.
- Our implementation is very efficient. In particular, we can compute the complete spatial self join of all the 1.3 billion geometric objects from the complete OpenStreetMap data (with a total size of 52.1 billion triples) in 1.5 hours on a standard PC.
- Our tool is open source and publicly available on GitHub, including documentation and instructions for how to reproduce the results from this paper.
- We provide an extensive evaluation on a variety of datasets. In particular, we show that in a fair comparison (with equal resources and optimal configuration for each) our code is over 500 times faster than the widely used PostgreSQL+PostGIS for self joins such as the above.
- We investigate in detail the effect of various speed-up heuristics. In particular, we explain why existing implementations like that of PostgreSQL+PostGIS are so comparatively slow, despite using similar core algorithms as we do.

2 RELATED WORK

The typical way to perform a spatial join, which we also use, consists of the following three phases: (1) retrieve all pairs of geometries, called *candidates*, where the bounding boxes intersect; (2) optionally reduce the candidate set further, using heuristics; (3) for each candidate, compute the exact value of the predicate. We will discuss the related work for each of these phases.

2.1 Candidate Retrieval

A good overview on algorithms for candidate generation can be found in the survey [10]. The survey focuses on intersection, whereas we consider all of: *contains*, *covers*, *within*, *intersects*, *touches*, *crosses*, and *equals*. The survey distinguishes between internal-memory methods (like R-tree indexing and other hierarchical data structures) and external-memory methods (like sweep-line techniques). The basic idea of sweep-line techniques is to sweep a vertical line over all bounding boxes, in order of ascending x-coordinate, and to maintain the set of rectangles that intersect the current sweep line. The typical data structure for that set is an interval tree. This technique can be used to determine whether a set of line segments intersect at all in time $O(n \log n)$ [16], and also to report all intersecting pairs in a set of axis-aligned rectangles [14]. Our approach builds on the latter, and is described in detail in Section 3.1.

2.2 Candidate Set Reduction

Brinkhoff and Kriegel [5] propose a set of simple geometric approximations to reduce the candidate set, like the rotated minimum bounding rectangle, the minimum bounding circle, or the convex hull. Another way of approximating geometries is by overlaying a grid over the geometric space and then representing a geometry by the grid cells which it intersects. Azevedo et al. [2] present a three-color covering, which for each geometry stores the set of cells which it fully covers and the set of cells which it partially covers. We describe and analyze a similar approach in Section 3.7. Georgiadis and Mamoulis [8] use a four-color scheme with the additional information whether a cell is covered more or less than 50%. They additionally enumerate the cells using the Hilbert curve, which allows for efficient compression and intersection at the same time. Google's *S2* library[3] combines the Hilbert-curve approach with a hierarchical grid that can be stored efficiently. S2 does not use colors, but explicitly maintains an outer covering (a set of cells which fully covers the geometry) as well as an inner covering (a set of cells which is fully covered by the geometry).

2.3 Exact Predicate Evaluation

For exact predicate evaluation, we again rely on a sweep-line algorithm. Bentley and Ottmann extend the algorithm by Shamos and Hoey to report *all* intersections in a set of line segments [4]. In this scenario, the algorithm cannot stop on the first intersection, and line segments may change their order relative to the sweep line at crossing points. To handle this, intersection points are added as future events, which requires a dynamic event list. This is typically realized via a priority queue, which results in a running time of $O((n + k) \cdot \log n)$, where k is the number of reported intersections. An important special case of this problem is the red/blue line-segment intersection problem, where two sets A and B of line segments are given, with the property that no two lines from A and no two lines from B have intersecting interiors. Then the relative order of the active segments in A and in B remains fixed during the line sweep. Mairson and Stolfi [12] proposed an $O(n \cdot \log n + k)$ algorithm for this problem, which was later simplified by Chan [6]. We use an approach similar to Chan's in Section 3.3.1.

2.4 Related Problems

Several works solve variations of our problem, by restricting the size of the input, the type of geometries, or the set of supported spatial predicates. Kipf et al [11] discuss the problem where a relatively small and static set of polygons (e.g., the streets and buildings of a city) are joined with a large set of points that are streamed into the algorithm (for example the current position drivers and potential passengers in a taxi app). They precompute an efficiently compressed quadtree-based index which stores a hierarchical, cell-based approximation of the polygons using Google's S2 library. In particular, they discuss applications where such an approximation is sufficient and the exact predicate evaluation can be omitted. Aghajarian et al. [1] use GPUs to solve the spatial join of polygons efficiently. Their approach is limited to intersection and requires that the complete input fits into the memory of the GPU. You et al. [18] discuss the problem of polyline intersection on GPU clusters.

2.5 Tools and Software

PostGIS[4] is a widely used extension for the *PostgreSQL* database that allows spatial joins with all the predicates that we also support. It supports the precomputation of R-tree indices for columns

[3]https://s2geometry.io
[4]https://postgis.net/

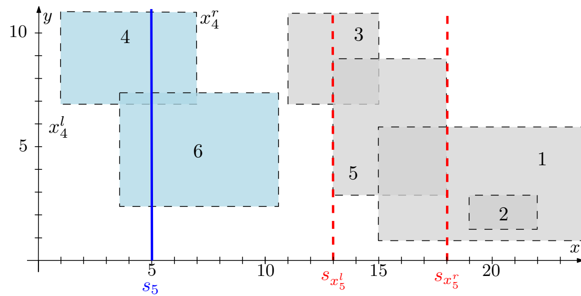

Figure 1: Reporting pairs of intersecting rectangles using a sweep line approach. For two rectangles to intersect, a vertical line must intersect both of them, We sweep such a line from left to right, checking each position. The set of rectangles intersecting s only changes at the left or right x coordinates x^l and x^r of rectangles.

with spatial data. When performing a spatial join, PostGIS only uses these indices for one of the inputs, even if indices for both inputs are present or a self-join is performed. For the exact predicate evaluations, PostGIS uses *libgeos*[5], a widely used library for geometric primitives. In Section 4, we compare our approach against PostgreSQL+PostGIS.

HyPerSpace [13] is the geospatial extension of the in-memory database *HyPer*; unfortunately, the code is proprietary. HyPerSpace relies on Google's S2 for the efficient processing of geospatial queries on datasets that change frequently.

OSCAR [3] allows specific spatial queries on the complete OSM data, namely finding all OSM objects in a given set of OSM regions. The core idea is to precompute a *cell arrangement* of all the given polygons. Then the mentioned spatial queries reduce to computing the intersection of lists of cell IDs. This approach can be considered as a variant of the grid approaches discussed in Section 2.2, making use of special properties of the OSM data (namely that the produced cells are relatively few and of even size).

3 APPROACH

Our basic method consists of the following two steps: (1) Candidate retrieval, which reports all geometries with intersecting bounding-boxes, and (2) Full geometric checks between candidates. We will first describe this baseline method, and gradually extend it with varying heuristics. The goal of all our heuristics is to prevent an expensive full geometric check between candidates. The effect of our techniques will then be evaluated in Section 4.

3.1 Candidate Retrieval

For the candidate retrieval, we are essentially given a set of axis-aligned bounding boxes $B(G) = ((x_i^l, y_i^l), (x_i^r, y_i^r))$ for all geometries G. Our goal is then to find all pairs of rectangles that intersect. As described above, we use a sweep line approach to produce a stream of candidate pairs (G_1, G_2).

The basic idea is that for two axis-aligned rectangles $B(G_1)$ and $B(G_2)$ to intersect, there must be a vertical line s_x intersecting both. We may thus sweep such a vertical line over the entire dataset and retrieve at all x positions the rectangles intersecting s_x. For each of these *active* rectangles, we then check whether their y-intervals also overlap. The active set only changes at the x^l and x^r positions of our rectangles. It is therefore sufficient to sweep along these x values. Figure 1 gives an example.

In the standard sweep line approach, all x_l and x_r and their corresponding object ID are stored as tuples (x_l, i, IN) and (x_r, i, OUT) in an *event* list E. Left x coordinates are an IN event, right x coordinates are an OUT event. If we now sort E by these x coordinates, an iteration over E is equivalent to sweeping s over the entire dataset, but skipping positions where the state does not change. On IN events, we add the corresponding rectangle id to an *active set* A. On OUT event, we remove the id from A.

At each event, we must now check whether the corresponding rectangle has an overlapping y-interval with any other rectangle in A. This can be done by using an *interval tree* for the active set. The operations *insert*, *delete*, and *lookup* on such an interval tree can be done in $O(\log n)$. Finding overlapping y-intervals for a single rectangle than takes time $O(\log n + k)$, where k is the number of overlapping intervals. This approach thus runs in $O(|O|\cdot(\log I+M))$, where M is the maximum number of rectangles intersecting a single rectangle, and I is the maximum size of A at any time.

Note that E can be easily held, sorted, and traversed on disk. We thus only require memory for the active set and the interval tree. For real-world input data, I is usually small (for our evaluation dataset OpenStreetMap, it was around 50,000).

3.2 Geometry Cache

Note that the candidate retrieval so far only delivered geometry IDs. For further checks, we need the actual geometric objects. As storing all full geometric objects in memory is unrealistic, we store them on disk and load them on demand. To avoid excessive loading of very large objects that are used very often in comparison (for example, think of the polygon of an entire country), we use a straightforward LRU cache. This has the effect that very large polygons requiring many comparisons usually remain in the cache until they are no longer required.

To further reduce both the I/O and the required disk space, we distinguish 5 different geometric types: **Points** are stored as raw 64 bit integer coordinates (x and y are 32 bits, respectively). **Simple Lines** are lines with only 2 anchor points, these are stored as 2 raw 64 bit integer coordinates. **Lines** that aren't simple are stored as pre-sorted lists of segments (this will be used in Section 3.3), together with their length, their bounding box and additional pre-computed information used by our heuristics (see below). **Simple Areas** are areas with less than 10 anchor points and without any holes, these are stored as raw lists of their 64 bit integer coordinates. **Areas** that aren't simple are stored as pre-sorted lists of segments, together with their holes (also pre-sorted), their bounding box, their geometric area, and again additional pre-computed information used by our heuristics. The basic idea of the distinction of "simple" and "normal" geometries is that for the simple geometries, the computation of the sorted list of segments, the bounding box and the various informations used by our heuristics is cheaper to do on the fly than loading it precomputed from disk.

[5] https://libgeos.org

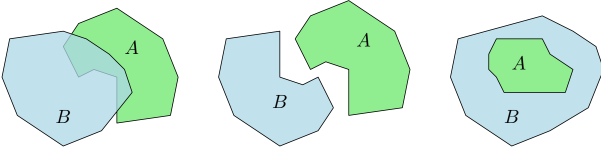

Figure 2: Two polygons have either intersecting boundaries, are disjoint, or one is completely contained in the other.

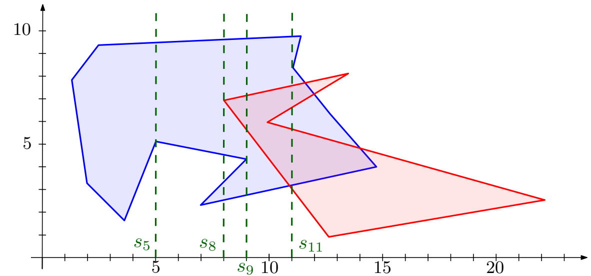

Figure 3: Sweeping over two sets of line segments (blue and red) to find intersections between blue/red and red/blue pairs. In this case, the line segments are polygon boundaries.

For our reference implementation, we used one cache per geometry type. The point cache had a maximum capacity of 1,000 objects (as it usually does not make much sense to keep points at all), all other caches could hold 10,000 objects.

3.3 Full Geometry Comparisons

To decide whether a candidate pair really fulfills the geometric relation, a full geometry comparison is necessary. We reduce all full geometry comparisons to two basic operations: (1) given two sets L_1 and L_2 of line segments, retrieve the set $C \subseteq L_1 \times L_2$ of all line segments that intersect each other, and (2) given a point P and a set L of line segments, find out whether P lies on any line segment of L or inside a closed polygon described by L.

It is easy to see that with these basic operations, we can decide whether a point is inside a polygon (Operation 2), whether a point intersects a line (Operation 2), or whether a line intersects another line (Operation 1). To check whether a line L_1 intersects a closed polygon described by L_2, we first use Operation 1 to decide whether L_1 intersects the polygon boundary L_2. If yes, we know that L_1 intersects L_2. If not, we must exclude the case where L_1 is completely inside the polygon described by L_2. This can be achieved by doing Operation 2 for a single random point of any segment in L_1. The same approach can be used for deciding whether a polygon intersects another polygon (see Figure 2 for an example). We note that for more complex comparisons (touches, covers, overlaps, crosses), a subsequent analysis of the intersecting line segments returned from Operation 1 is necessary. We will describe this at the end of this section.

Operation 2 is the standard point-in-polygon tests, which can for example be solved in linear time by a ray casting algorithm [17]. We will describe Operation 1 in more detail below.

Note that we ignore polygons with inner rings (holes) here for brevity, although our implementation supports them. In principle, if a polygon with a hole is involved in a comparison, the inner rings have to be compared separately to the other side, using the same approach as described here. To avoid iterating over all inner rings of polygons for each check (which might be problematic for polygons with a large number of such inner rings), we store for each polygon the bounding boxes of each inner ring, together with a list of inner ring IDs sorted be the leftmost x-coordinate of the ring. This enables us to quickly search for the first relevant inner ring, and to discard them as irrelevant based on the bounding box.

3.3.1 Sweep Line Approach. The naive approach for Operation 1 is a pairwise intersection test between all pairs $(l_1, l_2) \in L_1 \times L_2$, resulting in time $O(|L_1||L_2|)$. As mentioned above, a restricted variant of Operation 1 is known as the red/blue segment intersection problem: in this variant, the interiors of all line segments in A (and B respectively) are not allowed to intersect (that is, interior intersections are only allowed for pairs $(l_1, l_2) \in L_1 \times L_2$). This restriction can be easily fulfilled in our case: we simply pre-process all input geometries and add explicit anchor points at places where line segments intersect.

We first insert all line segments from L_1 and L_2 into a merged set L and mark each line segment as originating either from L_1 or L_2. We assume that for a line segment $l = ((x_1, y_1), (x_2, y_2))$, it already holds that $x_1 \leq x_2$ and use the more convenient notation $((x_l, y_l), (x_r, y_r))$. Next, we again build an event list E, into which each line segment l is inserted as two events: (x_l, o, l, IN) and (x_r, o, l, OUT), where $o \in 1, 2$ states whether l is from L_1, or L_2. E is again sorted by the left and right x coordinates.

As in the bounding box intersection case above, we require an active set that at each sweep line position s_x holds lines that are currently intersecting s_x, sorted by the y-coordinate of their intersection with s_x. For two line segments l and k, we say $l <_x k$ if the y-coordinate of the intersection of s_x with l is smaller than that of the intersection of s_x with k (we break ties in ambiguous cases, for example of an endpoint of l is exactly on k). We now exploit the fact that no two line segments from L_1, and no two line segments from L_2, have intersecting interiors. This means that when sweeping over the set L from left to right, the relation $<_x$ is *constant* for each pair $l_1, l'_1 \in L_1^2$, and each pair $l_2, l'_2 \in L_2^2$: no two line segments from the same set can ever "switch sides". We can thus maintain two *separate* active sets A_1 and A_2, and keep them correctly sorted according to $<_x$ using the following simple ordering relation: if a segment m was added to A before a segment n (if $x_{ml} < x_{nl}$), then $n < m$ exactly then if x_{nl} is to the right of m. If $x_{ml} > x_{nl}$, then $n < m$ exactly then if x_{ml} is to the left of n. If the left endpoint of n is on m (or vice versa), we use the right endpoint. If the left endpoint of n is exactly on the *right* endpoint of m, we break ties by always ordering $n < m$. In our implementation, we use a simple binary search tree (`std::set`) for A_1 and A_2. Figure 3 gives an example of such a line sweep.

Going over the event list E, we then have to consider the following cases:

(1) On event (x_l, o, l, IN): add x_l to A_o and search the other active set $A_{\bar{o}}$ for the line segments directly above and below of x_l (if they exist). In both directions, check whether x_l intersects with them and report if they do (e.g. at s_5 an s_8 in

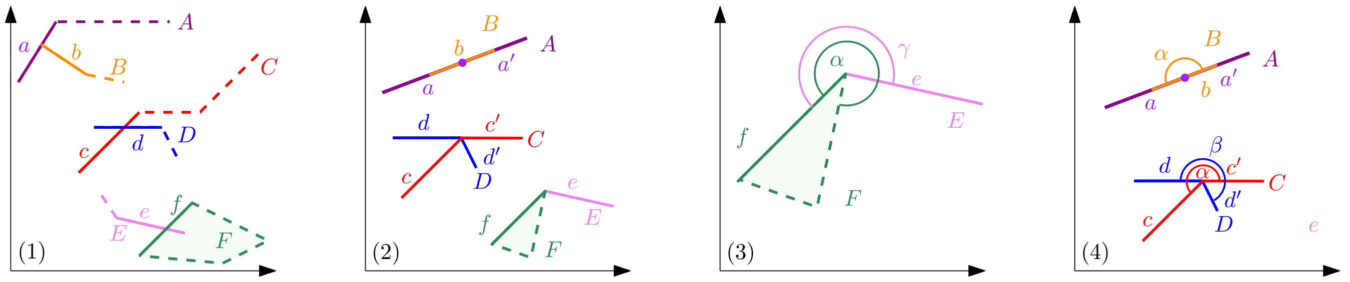

Figure 4: Using local segment intersections to decide global geometric relationships. (1): Trivial cases where a local intersection analysis is enough to decide global relationships. For example, a touches b locally, and A also touches B. e locally crosses into F, and E also is inside F. (2): Cases where local analysis is not enough: segment b both locally overlaps segments a and a', but line A covers line B. For C and D, we would locally decide that C and D touch, although they cross. e is right of f, but does not lie inside F, as another segment restricts the interior. (3) For each segmen, we store the outgoing angles of all adjacent segments. Then we can locally decide that e does indeed no cross into F by comparing α to γ. (4) Similarly, we can now decide that based on the comparison a vs. b, b does not overlap A, as the outgoing angle of the adjacent segment is exactly $180°$. For C and D, the presence of an outgoing angle at the intersections of d and c makes it clear that the local touch of d and c is not a global touch.

Figure 3). If there is an intersection in one direction (as is the case at $s_1 1$ in Figure 3), continue traversing and checking $A_{\bar{o}}$ in this direction until the first non-intersecting line segment is encountered. Note that the total number of these checks is bounded by $O(k)$, where k is the maximum number of intersections.

(2) On (x_l, o, l, OUT): remove x_l from A_o. Note that x_l might have been a "blocker" masking an intersection between a line segment from $A_{\bar{o}}$ directly below x_l and a line segment from A_o directly above x_l (or vice versa). This is for example the case at s_9 in Figure 3). We check all lines below x_l in $A_{\bar{o}}$ and all lines above x_l in A_o for pairwise intersections (and vice versa), again stopping in either direction as soon as we cannot find any more intersections. Again, the total number of these checks is bounded by $O(k)$.

Sorting the event lists takes time $O(|L| \log |L|)$, and sweeping over the events takes time $O(|L| \log \max(|L_1|, |L_2|)+k)$, resulting in a total running time of $O(|L| \log |L|+k)$. Note that $k \in O(|L_1| \cdot |L_2|)$.

3.3.2 Pre-Sorting Geometries. An important advantage of this approach is that we can directly store and pre-sort the event lists E_1 and E_2 for L_1 and L_2. For reporting the intersections between L_1 and L_2, we then do not have to materialize and sort the combined event list E: it is enough to iterate over the pre-sorted event lists E_1 and E_2 in a "zipper"-like fashion, producing the sorted event list E on the fly. (Another minor side effect is that we do not have to mark each event as originating from either L_1 or L_2.)

It is important to understand why this is relevant for our practical performance: theoretically, the difference between the bare sweep complexity $O(|L| \log \max(|L_1|, |L_2|) + k)$ and the total (including sorting) complexity $O(|L| \log |L| + k)$ does not appear to be very significant. However, the $O(\log \max(|L_1|, |L_2|))$ part comes from the lookup operations in the binary search trees holding the active sets. In practice, these sets are extremely small - for a convex polygon, they contain at most 2 segments, and in our testing datasets, the active set was typically smaller than 10. Also note that we have to do $k = |L_1| \cdot |L_2|$ additional iterations of the binary search tree in up and down direction only in the worst case - typically, k is very small. Even more importantly, the number of additional traversals per sweep line event is usually upper-bounded by a very small maximum size of the active sets. For almost all comparisons in our testing datasets, the sweeping could thus be assumed to run in linear time (and it strictly runs in linear time for comparisons between convex geometries), while the sorting of the merged event list E would have still required $O(|L| \log |L|)$ time. Additionally, we save the copying of E_1 and E_2 into a sorted merged list E. This is highly relevant in our scenario because we typically have millions of comparisons involving the same geometry - for example, in our testing dataset of the entire German OpenStreetMap data, we require (without any heuristic) a geometric check of all $\sim 84M$ houses in Germany against the geometry of Germany itself. Without the pre-sorting, we would have to sort the line segments constituting the German border 84 million times.

3.3.3 Analyzing Line Segment Intersections. If any two line segments l_1 and l_2 intersect, we would like to know the following: (1) do their geometries only touch at exterior bounds, or do they cross each other? (2) do they overlap? For checks against a part of a polygon boundary l_2, we would additionally like to know whether (3) l_1 lies on the inside our the outside of the polygon. This would enable us to compute all relations considered in this work (and to also fill the DE-9IM matrix) between the corresponding geometric objects (note that the case where no line segments intersect is trivial: either the corresponding geometries are disjoint, or one of the geometries is a polygon and completely contains the other).

Checking this seems straightforward: (1) and (2) are easy to check, and (3) is a matter of storing the polygon boundary l in its original clockwise orientation, in which case any intersecting line segment crossing to the right of l is on the "inside" side of the segment, and any line segment crossing to the left on the "outside" side. Figure 4.1 give examples. However, consider Figure 4.2. Here, we would for example decide that f crosses into the interior of the polygon corresponding to f - but this is not actually the case, as a neighboring line segment further restricts the interior of the polygon. A similar problem can be seen between C and D: here, we would locally decide that two lines touch each other, while in reality

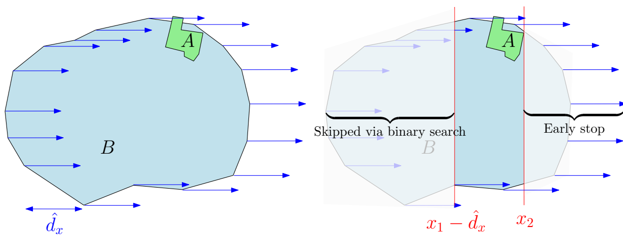

Figure 5: Comparing a large geometric B object to a smaller object A. In the list of sorted segments of B, we can skip all segments with $x_l < x_1 - \hat{d}_x$ using a simple binary search. When we reach a segment of B with $x_l > x_2$, we abort.

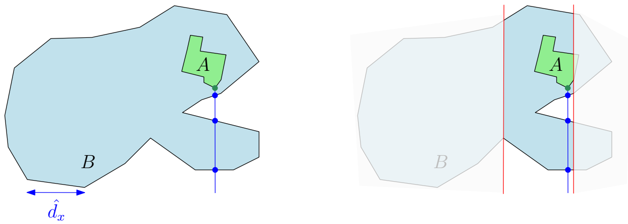

Figure 6: Checking whether a single random point of A is in B using raycasting with a ray $(x, -\infty)$ to (x, y). We can again search for the first segment of B with with $x_l >= x - \hat{d}_x$

they cross each other. To handle these problems, we additionally store for each line segment the left and right outgoing angle of the adjacent line segment as a *lookahead*. If the segment was the last (or first) segment of a line, we use a special placeholder value indicating "no adjacent segment". We can then avoid false local reporting by comparing angles, as shown in Figures 4.3 and 4.4.

3.4 Skipping Irrelevant Segments

The pre-sorting of line segments by their left x-coordinate enables other speed up heuristics. Consider Figure 5. It is easy to see that all line segments of B that end to the left of A are completely irrelevant for the sweepline process - they will have left the active set of B before any segment of A will become active. We may thus skip them completely by iterating to the first segment which is actually relevant for our process. If there are n segments to the left of A, this still requires iterating over n segments. As the list of segments is sorted, we would like to apply a simple binary search to get the number of iterations down to $O(\log n)$. But we cannot simply search for the first segment of B which ends after the first segment of A: the segments are sorted by the *left* x-coordinate. To nevertheless apply a binary search, we store for each geometric object the *longest* covered x-interval $\hat{d}_x$. Let x_a be the leftmost x-coordinate of A. Using binary search, we can then quickly search for the first segment of B with $x_l >= x_a - \hat{d}_B x$. We can also trivially abort our sweep line process as soon as we are past the rightmost segment of A, effectively restricting our sweep line process to a narrow band around A.

Note that the same technique can be used to speed up the point-in-polygon tests (Operation 2). Consider Figure 6. The standard ray-casting algorithm iterates over all line segments and checks for an intersection between a line going from infinity to $P = (x, y)$. If we chose this line to go from $(x, -\infty)$ to (x, y), we again can use a binary search to find the first segment of B with $x_l >= x - \hat{d}_B x$. If we then check the line segments for intersections with $((x, -\infty), (x, y))$ in their sorted order, we can again abort as soon as the first segment of B with $x_l > x$ appears.

3.5 Surface Area Precomputation

Recall that Operation 2 (Point-in-Polygon check) is required to differentiate between two cases after we have established that the segments of a geometry A and a polygon B do not intersect: (1) A is completely contained in B, or (2) A and B are disjoint. If both A and B are polygons, we have to both check if A is in B, and (if not), if B is in A. A simple preprocessing step to avoid one of these checks is to compute the surface area of all input polygons. Then, if the surface area of polygon A is smaller than the surface area of polygon B, we can already be certain that A cannot contain B.

3.6 Approximate Geometries

The techniques described so far aim to improve the performance of raw geometry comparisons. The reminder of this section will discuss heuristics to avoid such comparisons between full input geometries. Our techniques can be broadly classified into two categories: (1) *approximate geometries*, which are then again geometrically compared, and (2) a decomposition of input geometries using a static cell grid, which can then be used to decide some geometric relations without any geometric comparisons. We will first give a list of the simplified geometries used in our evaluation.

3.6.1 Diagonal Bounding Box. A bounding box simplification is already used in the geospatial index. This may be further refined by also computing *diagonal* bounding boxes - that is, the axis-aligned bounding box after the coordinate system has been rotated by 45°. The fixed orientation enables us to check for intersections using standard intersection tests for axis-aligned rectangles. Additionally, the boxes can be stored using only 2 coordinates. We store them directly in the event list for the candidate retrieval described in Section 3.1 (all other precomputations are stored in the cache). This heuristic may thus completely bypass the geometry cache.

3.6.2 Oriented Bounding Box. To improve on the diagonal bounding box, while still keeping the number of anchor points at only 4, we additionally precompute the oriented bounding box (OBB) of each polygon in O and use this OBB to quickly decide whether two geometries are disjoint.

3.6.3 Inner and Outer Ramer-Douglas-Peucker. To further improve the approximation, and to also allow us to quickly make positive contains or intersect decision based on the simplified geometries, we also precompute for each polygon A two approximations: a simplified *outer* polygon outer(A), and a simplified *inner* polygon inner(A). It obviously holds that $B \subseteq \text{inner}(A) \Rightarrow B \subset A$ and $B \not\subset \text{outer}(A) \Rightarrow B \not\subset A$, and also $G \cap \text{inner}(A) \neq \emptyset \Rightarrow G \cap A \neq \emptyset$ and $G \cap \text{outer}(A) = \emptyset \Rightarrow G \cap A = \emptyset$.

To compute outer(A) and inner(B), we modify the classic Ramer-Douglas-Peucker (RDP) algorithm for line simplification [7, 15]. Given a line L as an ordered list of anchor points, RDP takes an anchor point pair (p, u) (starting with the first and last point of L) and finds the point q between them with the largest distance

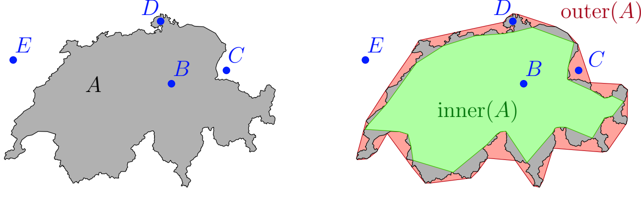

Figure 7: Using inner and outer simplified geometries for faster geometry comparison. If B is contained in the inner geometry, it is surely contained in A. If E is not contained in the outer geometry, it is surely disjoint with A.

$d = \text{dist}(q, p, u)$ to the line segment $\overline{pu}$. If d is smaller than a simplification threshold ϵ, all points between p and u are discarded. If $d > \epsilon$, q is kept, and the process recursively continues for (p, q) and (q, u).

Given a polygon A as a closed list of anchor points $(p_1, \ldots, p_n)$, we simplify the lines given by $(p_1, \ldots, p_{\lfloor n/2 \rfloor})$ and $(p_{\lfloor n/2 \rfloor+1}, \ldots, p_n)$ separately and later join the resulting simplified lines again. As a simplification criteria for RDP, we use the signed distance function

$$\text{dist}(q, p, u) = \frac{(x_u - x_p)(y_p - y_q) - (x_p - x_q)(y_u - y_p)}{\sqrt{(x_u - x_p)^2 = (y_u - y_p)^2}} \quad (1)$$

for a point $q = (x_q, y_q)$ and a line segment described by $p = (x_p, y_p)$ and $q = (x_q, y_q)$.

For the inner polygon, we keep q if $0 < \text{dist}(q, p, u) < \epsilon$. For the outer polygon, we keep q if $0 < -\text{dist}(q, p, u) < \epsilon$.

Note that for non-convex polygons, discarding a point to the right of a straight line segment does not necessarily mean that the resulting line segment is inside the original polygon. Similarly, discarding a point to the left does not mean that the resulting line segment is outside of the polygon. A simply mitigation strategy is to check a posteriori whether $inner(A)$ is really contained in A, and to discard $inner(A)$ otherwise. Similarly, if A is not contained in $outer(A)$, discard $outer(A)$.

During our experiments, we found that selecting a fixed simplification parameter for the inner and outer simplified polygons is problematic. Small simplification parameters will lead to little to no gains for large polygons. For smaller polygons, large simplification parameters will usually result in empty inner geometries, and an outer geometry that is equivalent to the convex hull. We settled on the following dynamic parameter: $\epsilon(A) = \alpha\sqrt{\text{area}(A)/\pi}$. This bases $\epsilon(A)$ directly on the (weighted) radius of a hypothetical circle with the same surface area as A. We set the weight $\alpha = 10$.

3.7 Intersecting Cell IDs

So far, the speed-up techniques still relied on geometric comparisons. This section describes a technique which allows to quickly decide whether a geometry G is definitely contained in a polygon P, whether it is definitely *not* contained in P, or whether it *might* be contained, without doing any geometric calculation (apart from the preprocessing). This technique is based on a grid covering the input dataset. The grid cells are continuously numbered from east to west, and north to south. We have found this approach to be more efficient than the more sophisticated hierarchical cell-covering from

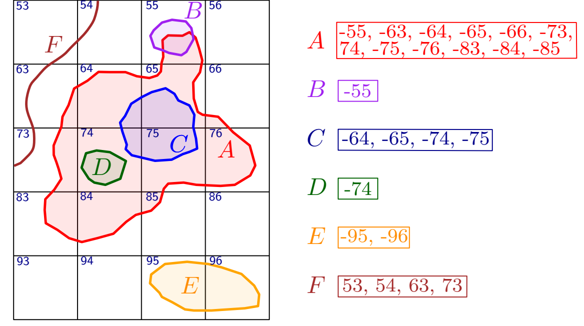

Figure 8: A static grid covering the bounding box of the entire dataset, with numbered cells. If a geometric object intersects a cell, it is added with a negative sign to its list of cell IDs. If a cell is completely contained in a polygon, the cell ID is added with a positive sign.

Google's *S2* library (see section 2.5), mostly because of the more expensive precomputation of the covering in S2.

3.7.1 Collecting Cell IDs. In a preprocessing step, we first compute for each polygon P the set A^+ of grid cells completely covered by P and the set A^- of grid cells only intersecting P. We define $A = A^+ \cup A^-$. The cells are represented by their integer id. Second, we compute for each line L the set A^- which holds all cells intersecting L (note that A^+ is naturally empty for lines). Note that we do not have to precompute cell IDs for points, as we can easily determine them on the fly. For notational convenience, we nevertheless also define a set A^- for points (it contains a single cell ID).

3.7.2 Geometric Contains and Intersect via Efficient List Intersection. Consider now for example two polygons P_1 and P_2 and their cell ID sets A_1^+, A_1^-, A_2^+, and A_2^-. If $|A_1 \cap A_2^+| = |A_1|$, then P_1 is surely completely contained in P_2. If $|A_1 \cap A_2| = |A^1|$, but $|A_1 \cap A_2^-)| \neq 0$, P_1 might be contained in P_2, requiring o a full predicate check.

Given two pair of cell ID sets (A_1^+, A_1^-) and (A_2^+, A_2^-) for two polygons P_1 and P_2. To capture the relations exemplified above, we determine the following measures: (1) $I^{\circ+} = |A_1 \cap A_2^+|$, (2) $I^{+-} = |A_1^+ \cap A_2^-|$, and (3) $I^{--} = |A_1^- \cap A_2^-|$. Table 1 gives an overview of how these numbers relate to geometric relations between pairs of geometries.

To compute these measures in a single pass, we combine A^+ and A^- in a list $L = (c_1, \ldots, c_n)$ in which cell IDs from A^+ are stored unchanged, and cell IDs from A^- are stored with a negative sign. L is then sorted by $|c_i|$. Computing $I^{\circ+}$, I^{+-} and I^{--} is then a list intersection problem between the two list L_1 and L_2, sorted by the (absolute!) values of their cell IDs. We compute this with an exponential search approach.

3.7.3 Efficient Calculation and Storage of Cell ID Lists. A naive calculation of the cell ID sets (A^+, A^-) for a polygon P would collect each cell c which intersects the bounding box of P, and check whether c is fully contained in P, or only intersects P. This would require a number of box-in-polygon checks which depends on the surface area of P, and might thus be quadratic in the worst case. To mitigate this, we scan P in a quadtree-like fashion. Let w be the side length of a grid cell, and let W and H be the width and height of the bounding box of p. We call grid cells of side length

Table 1: Relations between cell ID intersection measures and geometric comparisons. For each decision "yes" or "no", one of the conditions must be met. A — means that no decision is possible. No match requires a full predicate check.

	yes	no
intersects	$I^{\circ+} + I^{+-} > 0$	$I^{\circ+} + I^{+-} + I^{--} = 0$
contains	$I^{\circ+} = \|A_1\|$	$I^{\circ+} + I^{+-} + I^{--} \neq \|A_1\|$
covers	$I^{\circ+} = \|A_1\|$	$I^{\circ+} + I^{+-} + I^{--} \neq \|A_1\|$
touches	—	$I^{\circ+} + I^{+-} + I^{--} = 0$ $I^{\circ+} > 0$
overlaps	$0 < I^{\circ+} > \|A_1\|$	$I^{\circ+} + I^{+-} + I^{--} = 0$
crosses	—	$I^{\circ+} + I^{+-} + I^{--} = 0$

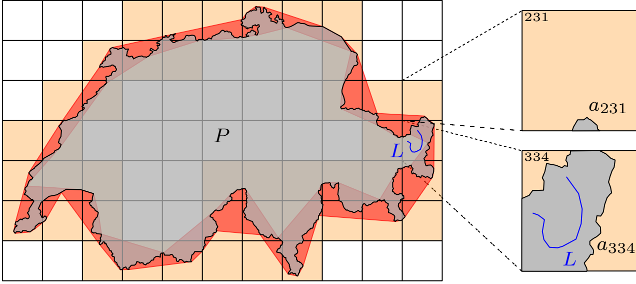

Figure 9: A line L and a polygon P sharing a single common cell ID (334), but the cell is not contained in P. To avoid a full geometry check between L and P, we store an offset to the first line segment of P in each cell.

w a *base cell*. Instead of scanning P with many small cells of width and height w, we begin with cells of width $\lceil W/w \rceil/4 \cdot w$ and height $\lceil H/w \rceil/4 \cdot w$. These larger *probe* cells are grid-aligned. Each of them exactly contains $\lceil W/w \rceil/4 \cdot \lceil H/w \rceil/4$ base cells. Let d be such a cell. Then, if d is fully contained in P, we add all contained base cells to A^+ and continue to the next cell. If d does not intersect with P, we skip it completely. If d intersects with P, but is not contained, we partition it into 4 smaller cells and check P against these cells. This process continues recursively. If d is a base cell, the recursion stops, and we add the corresponding base cell to A^-.

To avoid excessive memory consumption of the cell ID lists, we additionally use a simple running length encoding. Note that if a fully contained *probe* cell contains $n \cdot m$ base cells, it is then sufficient to simple add the cell with the lowest x value and a running length of n for each of the m rows.

3.8 Geometry Cutouts

We would also like to leverage the box IDs in cases where full geometric checks are still required. Consider Figure 9. If we know that line L and some polygon P only share a single cell ID (which P does not completely contain), it would be enough to check L against the part of P that is inside the cell. A naive way to allow such checks would be to store the geometric intersection between each non-contained cell and P, and to then check L only against the "cutout" geometry of the shared cell. See Figure 9, right for an example. However, storing these cutouts is expensive. A better way would be to store the segment intervals that lie inside each cell (in the example of Figure 9, one such interval would be required for box 231, and 2 intervals for box 334. Recall, however, that we store all geometries as lists of line segments, ordered by their left x-coordinate. The consecutive line segments inside a cell are not necessarily consecutive if sorted in this way. To nevertheless use a variant of these storage-friendly cutouts in our experiments, we store for each cell c a *single* offset to the first line segment that intersects c. In the example of Figure 9, we can then directly jump to the first line segment of P in cell 334 and start the the sweep line process there.

Table 2: Number of geometries in our four datasets, number of candidates for the self join, and total number of result pairs of all seven predicates.

	points	lines	polygons	#candidates	#results
OHM	5.2 M	1.8 M	1.2 M	311 M	734 M
FIN	1.5 M	3.2 M	4.2 M	211 M	351 B
GER	18.4 M	18.5 M	46.4 M	2.4 B	4.1 B
OSM	235 M	293 M	721 M	34.4 B	52.1 B

4 EXPERIMENTAL EVALUATION

We have implemented the approach described in Section 3 as a command-line tool called *spatialjoin*. It inputs two sets of geometries O_1 and O_2 and outputs a set of triples with all the geometric relations, as defined in Section 1.1. Specifically, the input is a single TSV file with one line per geometry and three columns: a unique ID for the geometry, an index indicating two which of the to input sets the geometry belongs (0 for O_1 and 1 for O_2), and a WKT representation of the geometry. The output is one line per triple. The code is publicly available on https://github.com/ad-freiburg/spatialjoin.

4.1 Setup

All our experiments were run on the same machine, with an AMD Ryzen 9 7950X processor with 16 physical and 32 virtual cores, 128 GB of RAM (DDR5), and 7.7 TB of disk space (NVMe SSD), running Ubuntu 22.04. For the PostgreSQL experiments, we use version 16.3 (with PostGIS version 3.4.2). For all its spatial joins, PostgreSQL uses one thread per index scan and one thread for the exact predicate evalatuations. For a fair comparison, we therefore run our tool with option *–num-threads 2*, which has the same effect. For the self-comparison of our tool with variants of itself or on different datasets, we use all the cores the machine provides.

4.2 Evaluation of Our Approach and Heuristics

We first evaluate the performance of our tool on *self joins*, the most expensive kinds of joins. In particular, we investigate the effect of the various heuristics for reducing the candidate set described in Section 3. We always compute the pairs for all seven predicates: *contains*, *covers*, *within*, *intersects*, *touches*, *crosses*, and *equals*.

We consider four datasets: all geometries from OpenStreetMap (OSM), the subset that lies in Germany (GER), the subset that lies in Finland (FIN), and all the geometries from OpenHistoricalMap (OHM). The dimensions of these datasets are shown in Table 2. FIN and OHM are of roughly the same size, but have very different

Table 3: Total time (in minutes) for computing the self-join with our tool, on each of our four datasets, for the seven heuristic settings described in Section 4.2. The speedup is relative to the setting shown in the last column.

	OHM		FIN		GER		OSM		
heuristics	time	speedup	time	speedup	time	speedup	time	speedup	relative to
bcsdoi	19.7 min		0.8 min		11.3 min		862 min		
Bcsdoi	12.6 min	1.57 ×	0.6 min	1.24 ×	7.0 min	1.61 ×	105 min	8.20 ×	bcsdoi
BCsdoi	12.8 min	0.99 ×	0.6 min	0.97 ×	6.8 min	1.04 ×	104 min	1.01 ×	Bcsdoi
BCSdoi	12.9 min	0.99 ×	0.7 min	0.97 ×	6.8 min	1.00 ×	107 min	0.97 ×	BCsdoi
BCSDoi	12.8 min	1.00 ×	0.6 min	1.07 ×	5.8 min	1.17 ×	94 min	1.14 ×	BCSdoi
BCSdOi	12.7 min	1.01 ×	0.7 min	0.98 ×	6.8 min	0.99 ×	105 min	1.01 ×	BCSdoi
BCSdoI	13.8 min	0.93 ×	0.7 min	1.01 ×	6.6 min	1.02 ×	107 min	0.99 ×	BCSdoi

characteristics: OHM has many similar regions, which makes self-joins significantly harder (because no heuristic we know of can filter out a pair of two very similar objects).

For each dataset, we evaluated the effect of each of our seven heuristics from Section 3:

- **B** cell IDs, also called box IDs (Section 3.7)
- **C** cutouts (Section 3.8)
- **S** precomputed surface area (Section 3.5)
- **D** diagonal bounding box (Section 3.6.1)
- **O** oriented bounding box (Section 3.6.2)
- **I** inner/outer simplified geometries (Section 3.6.3)

Each of these six heuristics can be switched on or off. We evaluated all of the resulting 64 combinations. We found that all the significant effects can be observed via the relative speed-ups of these seven combinations: `bcsdoi`, `Bcsdoi`, `BCsdoi`, `BCSdoi`, `BCSDoi`, `BCSdOi`, `BCSdoI`. Each combination is represented by a six-letter string, where an uppercase letter means that the heuristic is enabled, and a lowercase letter means that the heuristic is disabled. The first combination is the baseline, with all heuristics disabled. The next three combinations successively add one of the heuristics `B`, `C`, and `S`. The last three combinations add *either* of the three approximation heuristics `D`, `O`, or `I` from Section 3.6.

Table 3 reports the *total* running time of our tool for self joins of each of the four datasets and for each of the seven combinations. The most effective heuristic is the cell IDs, especially for the very large OSM dataset. This is understandable as cell IDs are particularly effective when they save an exact predicate evaluation between a small geometry (like a building) and a large geometry (like a whole city). The effect is larger for OHM than for FIN (despite their similar size) because OHM has many more points and fewer polygons; see Table 2. The second most effective heuristic are the diagonal bounding boxes. Of the three approximate-geometry heuristics, they yield the best trade-off between quality of approximation, cost of computation, and storage cost. As the diagonal bounding boxes are stored directly in the event list of the sweep, they also avoid costly geometry cache loads for candidate false positives. They don't improve performance relative to the cell IDs for OHM, which has many polygons that differ only slightly at the boundary; then no heuristic can save the expensive predicate evaluations.

The effect of `C` was marginal. The `S` heuristic gave hardly any speedup at all. We found that the main reason for this is because the single point-in-polygon test that it saves is very cheap.

The effect of adding the inner/outer geometries (`I`) was also insignificant. We assume that this is mainly because of two effects: (1) the bounding box and diagonal bounding box filter during the candidate reterieval form an implicit simplified outer polygon that is often very close to the outer simplified geometry, and (2) the cases where `I` would have been effective are exactly the cases where the cell IDs (`C`) are effective - comparisons between very large and very small geometries. Also note that for geometries that are nearly equivalent, `I` adds an additional geometric test between the simplified geometries, but will have no filtering effect. This additional but ineffective work is noticable and explains the significantly slower running times with `I` on the OHM dataset.

4.3 Comparison to PostgreSQL+PostGIS

We compare the running time of our tool to PostgreSQL (with the PostGIS extension, which gives PostgreSQL its spatial-join capabilities). We consider both self joins and other spatial joins. For PostgreSQL, we load each dataset into a separate table and build a spatial index over the geometry column. We do not measure the time this takes for PostgreSQL, and for a fair comparison, we also do not measure the corresponding time our tool takes (whereas in Section 4.2 we consider the total time, including the parsing of the input file). For all queries, we separately measured the time for only the candidate generation (in PostgreSQL this can be achieved via the && operator, our tool has an option *–no-geometry-checks* for this) and the time for computing the full spatial join (including the time for the candidate generation). For PostgreSQL, we use the *ST_Intersects* predicate while our tool always computes all of the spatial predicates. We also tried the other predicates for PostgreSQL, but found them to be impractically slow.

4.3.1 Self joins. Table 4 shows how our tool compares with PostgreSQL regarding self joins. As explained in Section 4.1, we ran our tool with option *–num-threads 2* for a fair comparison with PostgreSQL; the running times are therefore slower than those in Table 3. We used the best combination of heuristics, which is `BCSDoi`. We see an enormous difference in running time. While our tool can compute the self-join even for the huge OSM dataset (with 1.3 B geometries, see Table 2) in less than 10 hours, PostgreSQL does not finish within 10 hours (after which we aborted it) even for the small FIN dataset (with 8.9 M geometries). Because of this large discrepancy, we also ran PostgreSQL on an even smaller dataset (all OSM geometries in Berlin, with 2.1 M geometries), where it took

Table 4: Comparison of our tool with PostgreSQL for self joins, on all four datasets. The table shows both the total computation time and the time for candidate generation (which was measured separately). If the computation was not finished after 10 hours, it was aborted.

	PostgreSQL		Ours	
	candidates	intersects	candidates	all predicates
OHM	10.7 min	> 10 h	5.9 s	41.7 min
FIN	27.0 min	> 10 h	6.8 s	30.6 s
GER	> 10 h	> 10 h	1.7 min	21.0 min
OSM	> 10 h	> 10 h	25.1 min	9.2 h

6.4 hours to compute the self join, versus 42 seconds for our tool (which is more than 500 times faster). The table also shows the time used for the candidate generation. For the two smallest datasets (OHM and FIN), PostgreSQL can compute the candidate set in less than an hour, but still two orders of magnitude slower than our tool. For the two larger datasets, even candidate generation is out of reach for PostgreSQL in a reasonable time. This shows that for self joins, both candidate generation and exact predicate evaluation are a bottleneck for PostgreSQL, though the bulk of the performance is lost in the latter.

We investigated the reasons for PosgreSQL's poor performance by inspecting its query plans and source code. The candidate generation is slow because PostgreSQL only uses its R-Tree index for one side of the (self-join) and performs one index lookup for each geometry in the other side. This is much less efficient than our sweep-line approach for joins where both sides are large. The exact predicate evaluations are slower because PostgreSQL does not use filtering heuristics and because it always has to sort the line segments of each geometry by x-coordinate before being able to perform the actual predicate evaluation, whereas we store this sorted representation as part of our index structure (see Section 3.3.2). This saves a lot of time when each geometry is involved in many checks, which is typical for self joins.

4.3.2 Other Joins. We computed the following non-self spatial joins with both PostgreSQL+PostGIS (again, only `ST_Intersects`) and our tool (again, all predicates). The geometries are again from OpenStreetMap. Apart from the queries, the setup was the same as in the previous section. The results can be seen in Table 5.

Q1 1.1 M restaurants ⋈ 1.3 M transit stops[6]
Q2 66.4 M residential streets ⋈ 0.7 M administrative regions
Q3 66.4 M residential streets ⋈ 66.4 M residential streets
Q4 0.9 M powerlines ⋈ 66.4 M residential streets

For Q1, which involves only simple axis-aligned rectangles, the candidate generation of our tool is about 10 times faster than that of PostgreSQL. For Q3 and Q4, which mainly involve simple line geometries, it is about 35 times faster. For Q2, the number of candidates per element on the right side is largest because administrative regions typically contain thousands of streets. For this query, the candidate generation of our approach was nearly 70 times faster than that of PostgreSQL.

[6]Using axis-aligned squares with a side length of 1km around the point geometries on both sides.

Table 5: Running times for the four non-self spatial-join queries described in Section 4.3.2. As in Table 2, the column #results gives the total number of results for all seven predicates. As in Table 4, the time only for candidate generation and the total time are shown separately.

		PostgreSQL		Ours	
	#results	candidates	intersects	candidates	all predicates
Q1	19.3 M	6.8 s	12.5 s	0.7 s	10.6 s
Q2	279.2 M	21.7 min	> 10 h	19.0 s	5.6 min
Q3	230.3 M	8.9 min	12.5 min	31.5 s	7.2 min
Q4	1.3 M	6.5 min	10.6 min	12.5 s	1.1 min

For queries involving only points, lines and/or simple polygons on both sides (Q1, Q2, and Q4), PostgreSQL spent most of its time for candidate generation. For example, for the simple query Q1, the exact predicate check only took an additional 5.7 seconds, while our approach took an additional 9.9 seconds. For Q2, which involves many checks between small streets and very large polygons, we aborted the full `ST_Intersects` query after 10 hours, while our approach took only 5.6 minutes.

This is consistent with our findings in the previous section: The exact predicate evaluation of PostgreSQL is efficient when the involved geometries are simple, or when each geometry is involved in only few candidate pairs. Both of those hold for Q1, Q3, and Q4, but not for Q2 where the complex but few polygons of administrative regions are involved in many predicate checks. Our tool efficiently handles all of those cases.

5 CONCLUSIONS AND FUTURE WORK

We have described an efficient algorithm and implementation for computing spatial joins between very large sets of geometric obects in 2D. Our code is publicly available on GitHub, with instructions for how to reproduce the results from this paper. Our code is significantly faster than the widely used PostgreSQL+PostGIS in all cases, and orders of magnitude faster when the number of candidate pairs is large or many complex geometries are involved. In particular, this is often the case for self joins.

We found that there are three main ingredients for this performance difference, which we all consider but PostgreSQL misses. First, when both sets are large, it is important to use a spatial index for both sides (PostgreSQL always iterates over one side). Second, when a spatial join involves many predicate evaluations with the same complex geometry, a suitable pre-processing of that geometry is crucial (which PostgreSQL does not do). Third, cell IDs are an invaluable heuristic when comparing small with large geometries, and diagonal bounding boxes provide another significant performance boost at almost no cost (PostgreSQL employs no such heuristics).

We currently provide a command-line tool to compute spatial joins. We consider developing this into a C++ library, which eventually may even serve as a drop-in replacement for *libgeos*.

REFERENCES

[1] Danial Aghajarian, Satish Puri, and Sushil K. Prasad. 2016. GCMF: an efficient end-to-end spatial join system over large polygonal datasets on GPGPU platform. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*, Siva Ravada, Mohammed Eunus Ali, Shawn D. Newsam, Matthias Renz, and Goce Trajcevski (Eds.). ACM, 18:1–18:10. https://doi.org/10.1145/2996913.2996982

[2] Leonardo Guerreiro Azevedo, Ralf Hartmut Güting, Rafael Brand Rodrigues, Geraldo Zimbrão, and Jano Moreira de Souza. 2006. Filtering with raster signatures. In *14th ACM International Symposium on Geographic Information Systems, ACM-GIS 2006, November 10-11, 2006, Arlington, Virginia, USA, Proceedings*, Rolf A. de By and Silvia Nittel (Eds.). ACM, 187–194. https://doi.org/10.1145/1183471.1183503

[3] Daniel Bahrdt and Stefan Funke. 2015. OSCAR: OpenStreetMap Planet at Your Fingertips via OSm Cell ARrangements. In *WISE (1) (Lecture Notes in Computer Science, Vol. 9418)*. Springer, 153–168. https://link.springer.com/chapter/10.1007/978-3-319-26190-4_11

[4] Jon Louis Bentley and Thomas Ottmann. 1979. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Trans. Computers* 28, 9 (1979), 643–647. https://doi.org/10.1109/TC.1979.1675432

[5] Thomas Brinkhoff and Hans-Peter Kriegel. 1994. Approximations for a Multi-Step Processing of Spatial Joins. In *IGIS '94: Geographic Information Systems, International Workshop on Advanced Information Systems, Monte Verita, Ascona, Switzerland, February 28 - March 4, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 884)*, Jürg Nievergelt, Thomas Roos, Hans-Jörg Schek, and Peter Widmayer (Eds.). Springer, 25–34. https://doi.org/10.1007/3-540-58795-0_31

[6] Timothy M. Chan. 1994. A Simple Trapezoid Sweep Algorithm for Reporting Red/Blue Segment Intersections. In *Proceedings of the 6th Canadian Conference on Computational Geometry, Saskatoon, SK, Canada, August 1994*. University of Saskatchewan, 263–268.

[7] David H Douglas and Thomas K Peucker. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization* 10, 2 (1973), 112–122.

[8] Thanasis Georgiadis and Nikos Mamoulis. 2023. Raster Intervals: An Approximation Technique for Polygon Intersection Joins. *Proc. ACM Manag. Data* 1, 1 (2023), 36:1–36:18. https://doi.org/10.1145/3588716

[9] Herring et al. 2011. OpenGIS Implementation Specification for Geographic information – Simple feature access – Part 1: Common architecture. (2011). https://www.ogc.org/standard/sfa/

[10] Edwin H. Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Trans. Database Syst.* 32, 1 (2007), 7. http://www.cs.umd.edu/~hjs/pubs/jacoxtods07.pdf

[11] Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2018. Adaptive Geospatial Joins for Modern Hardware. *CoRR* abs/1802.09488 (2018). arXiv:1802.09488 http://arxiv.org/abs/1802.09488

[12] Harry G. Mairson and Jorge Stolfi. 1988. Reporting and Counting Intersections Between Two Sets of Line Segments. In *Theoretical Foundations of Computer Graphics and CAD*, Rae A. Earnshaw (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 307–325.

[13] Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. 2016. High-Performance Geospatial Analytics in HyPerSpace. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 2145–2148. https://doi.org/10.1145/2882903.2899412

[14] Franco P. Preparata and Michael Ian Shamos. 1985. *Computational Geometry - An Introduction*. Springer. https://doi.org/10.1007/978-1-4612-1098-6

[15] Urs Ramer. 1972. An iterative procedure for the polygonal approximation of plane curves. *Comput. Graph. Image Process.* 1, 3 (1972), 244–256. https://doi.org/10.1016/S0146-664X(72)80017-0

[16] Michael Ian Shamos and Dan Hoey. 1976. Geometric Intersection Problems. In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*. IEEE Computer Society, 208–215. https://doi.org/10.1109/SFCS.1976.16

[17] M. Shimrat. 1962. Algorithm 112: Position of point relative to polygon. *Commun. ACM* 5, 8 (1962), 434. https://doi.org/10.1145/368637.368653

[18] Simin You, Jianting Zhang, and Le Gruenwald. 2016. High-performance polyline intersection based spatial join on GPU-accelerated clusters. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2016, Burlingame, California, USA, October 31, 2016*, Varun Chandola and Ranga Raju Vatsavai (Eds.). ACM, 42–49. https://doi.org/10.1145/3006386.3006390