Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins

Andreas Kipf Harald Lang Varun Pandey Raul Alexandru Persa
Christoph Anneser Eleni Tzirita Zacharatou[*] Harish Doraiswamy[◇]
Peter Boncz[★] Thomas Neumann Alfons Kemper
TUM TU Berlin[*] NYU[◇] CWI[★]
{kipf, langh, pandey, raul.persa, anneser, neumann, kemper}@in.tum.de
eleni.tziritazacharatou@tu-berlin.de harishd@nyu.edu boncz@cwi.nl

ABSTRACT

Connected mobility applications rely heavily on geospatial joins that associate point data, such as locations of Uber cars, to static polygonal regions, such as city neighborhoods. These joins typically involve expensive geometric computations, which makes it hard to provide an interactive user experience.

In this paper, we propose an adaptive polygon index that leverages *true hit filtering* to avoid expensive geometric computations in most cases. In particular, our approach closely approximates polygons by combining quadtrees with true hit filtering, and stores these approximations in a query-efficient radix tree. Based on this index, we introduce two geospatial join algorithms: an approximate one that guarantees a user-defined precision, and an exact one that adapts to the expected point distribution. In summary, our technique outperforms existing CPU-based joins by up to two orders of magnitude and is competitive with state-of-the-art GPU implementations.

1 INTRODUCTION

Connected mobility companies need to process vast amounts of location data in near real-time to run their businesses. For example, Uber needs to map locations of cars and passenger requests (points) to predefined zones (polygonal regions) for allocation and dynamic pricing purposes [40]. These polygonal regions are typically largely disjoint (non-overlapping) and mostly static. Points, on the other hand, are often not known a priori. Thus, the problem is how to efficiently find the polygons that contain an incoming point.

Traditionally, such point-polygon joins [19] follow the *filter and refine* approach. In this two-phase evaluation strategy, the filtering phase typically uses an index (e.g., an R-tree) on the minimum bounding rectangles (MBRs) of polygons and probes the index for each point to obtain a list of candidate join pairs. Then, in the refinement phase, expensive point-in-polygon (PIP) tests are performed to discard false matches.

We argue that the time has come to rethink this strategy: First, main memory is not a scarce resource anymore and modern machines offer multiple terabytes of memory. Combined with the city-centric model of geospatial applications (e.g., Uber), we show that it is possible to maintain highly fine-grained indexes for entire cities (e.g., Uber's operating zones) in main memory, dramatically reducing the number of CPU-intensive PIP tests. Second, geospatial positions, nowadays typically obtained by smartphones or wearables, are inherently imprecise [41]. Thus, we argue that it is in many cases admissible to trade off accuracy for performance. Based on these two insights, we transform the traditionally CPU-intensive problem of point-polygon joins into one that is bound by memory access latencies.

In contrast to the classical filter and refine approach, *true hit filtering* [9] identifies actual join pairs already in the filtering phase, and thus partially avoids expensive refinements. This is achieved by using additional approximations (such as inner rectangles [20]) to approximate the interior of polygons, so that when a point falls into an interior approximation, it can be safely deducted that the point is contained in the polygon.

Building on this seminal idea, we present an improved algorithm that combines true hit filtering with quadtrees [23] to *holistically* index an entire set of polygons. This is in contrast to existing implementations of true hit filtering that approximate polygons individually [15, 21] or use non-hierarchical (single-resolution) grids [6, 39, 49]. In our approach, polygons are translated into a *single* set of multi-resolution grid cells that approximates their boundary and interior areas. To support efficient queries, we store one-dimensional identifiers of the cells in a new in-memory radix tree (trie) named *Adaptive Cell Trie* (ACT). We show that ACT is more query-efficient than previous approaches for indexing cell identifiers (e.g., B-trees, like in [21]).

Another distinguishing feature of our approach is that it can entirely avoid the expensive refinement phase by refining cells in the boundary areas until a user-defined precision is guaranteed. Naturally, this comes at the cost of higher memory consumption than traditional filter and refine approaches. However, as stated above, we argue that we can nowadays actually afford this higher memory consumption in exchange for higher performance.

Our approach can also provide accurate results by performing expensive PIP tests for points that are potential hits. To reduce their number, we adapt (train) our index based on historical data points to provide higher precision where it is actually needed. As we show in our experiments, our accurate algorithm performs very few PIP tests. Compared to a filter based on the polygons' MBRs, our index (trained with 1 M historical points) reduces the number of required PIP tests by >97% for a join between NYC taxi pick-up locations and neighborhood polygons. This algorithm can also be used when ACT cannot guarantee the desired precision given a certain memory budget.

In summary, we make the following contributions:

- An algorithm that computes quadtree-based grid approximations for sets of polygons with precision guarantees
- A radix tree data structure (ACT) that is optimized for indexing cell identifiers: for a join of NYC's yellow taxi data with NYC's neighborhoods, we achieve a throughput of >50 M points/s per CPU core under a <4 m precision bound

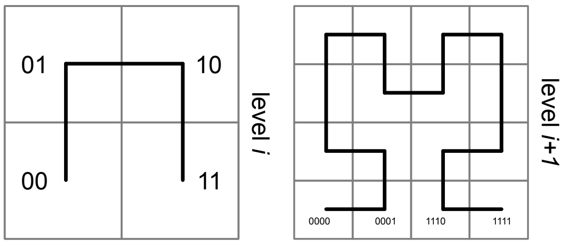


Figure 1: Quadtree-based cell decomposition and Hilbert curve-based enumeration.

- An evaluation of ACT in contrast to more traditional data structures, such as B-trees
- An accurate algorithm that trains the index structure based on historical data points
- An experimental comparison against state-of-the-art GPU-based point-polygon joins

In the remainder of this paper, we first give some background about the building blocks of our approach in Section 2. Section 3 describes our approach and Section 4 presents the evaluation with real-world and synthetic data. Finally, we summarize related work in Section 5 before concluding in Section 6.

2 BACKGROUND

Location Discretization. Our approach relies on a quadtree-based (hierarchical) decomposition of space (the surface of the Earth in this case). This decomposition is static and thus *data independent*. We enumerate the quadtree cells using a space-filling curve (e.g., the Hilbert or the Z curve) to index them in a one-dimensional data structure. Our approach does not depend on a concrete space-filling curve. For our indexing strategy to work, the cell enumeration must only fulfill the property that child cells share a common prefix with their parent cell.

Figure 1 shows the hierarchical decomposition of two cells at levels i and $i+1$ and the corresponding bitwise representations that encode the cells' positions along the Hilbert curve. Each cell consists of four sub cells, which it completely covers. Child cells share a common prefix with their parent cell, allowing us to compute *contains* relationships using efficient bitwise operations. In our implementation, we use the Google S2 library [32] for mapping latitude/longitude coordinates to 64-bit cell identifiers, which we call *cell ids* in the following. A cell id encodes up to 30 levels with two bits per level.

Polygon Approximations. To obtain fine-grained polygonal approximations, we need a method that maps polygons to sets of quadtree cells (possibly at different levels). In particular, our algorithms take as input approximations of the boundary and interior areas of *single* polygons. In our implementation, we use the S2 library to obtain these individual polygon approximations. Figure 2 illustrates a covering (in blue) and an interior covering (in green) of a polygon. A point contained in a covering cell is either within or outside of the polygon while points that match interior cells are known to be within the polygon (true hits). The cell marked with 1 is one of the largest covering cells and only minimally intersects the polygon. Any point contained in this cell has at most a distance of $\sqrt{2} * \epsilon$ (with ϵ being the side length of the cell) to the polygon. To allow for an efficient search, S2 stores the cell ids of a covering in a sorted vector. Besides sorting the cell id vector, it allows for the covering to be *normalized*. A normalized covering contains neither conflicting nor duplicate cells. Two cells are conflicting when one cell contains the other. Only when the covering is normalized can cell containment checks be efficiently implemented using a binary search on the sorted vector ($O(\log n)$). While binary search on a sorted vector is a good strategy for querying small collections of cells (e.g., the covering cells of a single polygon), it is not the most efficient way to search larger collections (e.g., coverings of multiple polygons). In this work, we store large cell collections in ACT, a query-efficient radix tree, and evaluate its performance compared to alternative physical representations (including a sorted vector and a B-tree).

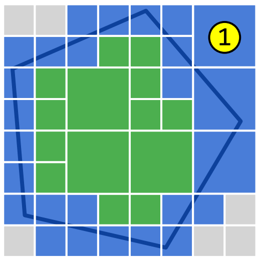


Figure 2: A covering (blue cells) and an interior covering (green cells) of an individual polygon.

PIP Test. A point-in-polygon (PIP) test determines whether a point lies within a polygon. Typically such a test is performed using complex geometric operations, such as the *ray-tracing algorithm* [17], which involves drawing a line from the query point to a point that is known to be outside of the polygon and counting the number of edges that the line crosses. If the line crosses an odd number of edges, the query point lies within the polygon. The runtime complexity of this algorithm is $O(n)$, n being the number of edges. While there are many conceptual optimizations to the PIP test, this operation remains computationally expensive since it processes real numbers (e.g., latitude/longitude coordinates) and thus involves floating point arithmetics.

3 GEOSPATIAL JOIN APPROACH

In this work, we target the problem of mapping points to static, largely disjoint polygons. We show how to accelerate such joins by computing fine-grained cell-based approximations of sets of polygons and maintaining them in a query-efficient in-memory radix tree, which enables efficient cell lookups and significantly reduces (or even eliminates) expensive geometric tests.

In contrast to techniques that first reduce the number of candidate polygons using an index, e.g., an R-tree on the polygons' MBRs, and then refine candidates using geometric operations, our approach leverages true hit filtering [9] and identifies most or even all join pairs in the filter phase. On a high level, our approach first computes cell-based approximations of all polygons, called coverings and interior coverings, and merges them to form a *super covering*. Then, it stores these approximations in a specialized in-memory radix tree (named ACT) which allows for efficient lookups. Finally, ACT is probed for every point to obtain a list of *true* and *candidate* point-polygon pairs. The candidate pairs are either refined by performing geometric computations to obtain an accurate result, or deemed to be part of the join result when small approximation errors can be tolerated.

The following provides more information about our indexing technique and the two geospatial join algorithms that are based on it: the approximate one that completely avoids expensive PIP tests while still guaranteeing a user-defined precision, and the exact one that reduces expensive computations by adapting to the expected point distribution. These algorithms allow us to trade memory consumption with precision (approximate approach) and performance (exact approach). Thus, they both favor

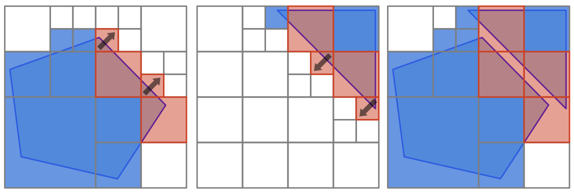

(a) Covering (b) Covering (c) Combined cov.

Figure 3: A combined covering may be less selective than two individual coverings. The arrows indicate that the cells will be expanded.

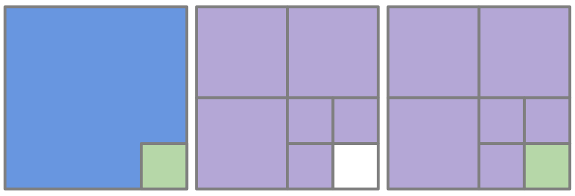

(a) c_1 and c_2 (b) Difference d (c) d and c_2

Figure 4: Precision preserving conflict resolution. c_1 is marked in blue, c_2 in green, and the cells in d in purple. Note that c_1 contains c_2.

modern hardware with large main memory capacities and high memory bandwidths. In summary, the key contribution of our indexing strategy is the novel combination of the super covering that approximates the polygons precisely, and the radix tree data structure that allows these approximations to be queried efficiently. With this design, we revisit the concept of true hit filtering in the context of modern hardware.

3.1 Adaptive Cell Trie (ACT) Indexing

3.1.1 Super Covering Computation. The super covering consists of a set of multi-resolution grid cells. All grid cells are disjoint in the sense that each geographical point is covered by at most one cell, even if two (or more) polygons overlap. A single cell of the super covering can therefore be associated with multiple polygons. The super covering maintains a list of *polygon references* for each individual cell. A polygon reference has two attributes:

polygon id The identifier of the polygon that this cell references.

interior flag Whether the cell is an interior or a boundary cell of the polygon.

The precision of the super covering determines the selectivity of the index. When combining the approximations of the individual polygons, we need to take special care of conflicting cells[1] to not lose precision. However, this is challenging for two reasons. First, conflicts may occur between the cells of a covering of a given polygon and the cells of its interior covering. The interior cells always overlap some (if not all) covering cells. Second, when different polygons overlap or are close to each other, conflicts may occur between the cells of their coverings.

One approach for retaining the precision would be to not resolve such conflicts at all and maintain all conflicting cells. However, this would have the consequence that a query point could match with more than one cell, which would affect lookup performance. In a radix tree, this would mean that we would need to keep searching lower levels once we found a match at a higher level.

Ensuring that cells are non-overlapping results not only in higher lookup performance, but also in a more compact radix tree. The reason is that for a given entry in a tree node, we only need to differentiate between a pointer (to a child node) and a value. With overlapping cells, we would have to store a pointer *and* a value.

There are two obvious solutions for resolving a conflict between two cells c_1 and c_2, where c_1 is an ancestor of c_2 in the quadtree (c_1 contains c_2). One is to replace c_2 with c_1, which leads to a precision loss as shown in Figure 3. Figures 3a and 3b show the coverings of two individual polygons. The red cells have conflicts with cells of the other covering. Figure 3c shows a combined covering, where the originally smaller cells are subsumed by larger cells, causing a precision loss. The other solution is to replace c_1 with a set of smaller cells at the same level (i.e., of the same size) as c_2. While this retains the precision, it can significantly increase the number of cells in the combined covering.

Without compromising on precision, we would like to reduce the number of cells introduced. To solve this problem, instead of storing both conflicting cells c_1 and c_2, we compute their difference d and store c_2 and d. This has the advantage that there will not be any overlap between the indexed cells, and thus an index lookup will return *at most* a single cell. The side effect is that the total number of cells will increase since d consists of at least three cells.

Figure 4 illustrates this precision preserving conflict resolution. Assume that c_1 (blue) and c_2 (green) are cells of two different coverings and that c_1 contains c_2. First, we compute d, which consists of six cells. We then copy all polygon references of c_1 to d and c_2 and omit c_1. Note that the cell count is increased by five. Overall, our approach retains the precision and the type (boundary or interior) of the individual cells as well as the mappings of cells to polygons.

Listing 1 outlines this algorithm. We iterate over all input cells and try to insert them into the super covering. When a cell already exists, this means that it is also part of another covering that has already been processed. When two cells conflict, this means that either the current cell covers the other cell or vice versa. These two cases may happen when polygons overlap or are close to each other, or when we first insert the cells of the covering of a given polygon and then the cells of its interior covering. To address these cases, we apply the precision preserving conflict resolution strategy described above. As mentioned earlier, this strategy increases the total number of cells. However, a more precise index reduces the number of expensive PIP tests and thus increases overall performance.

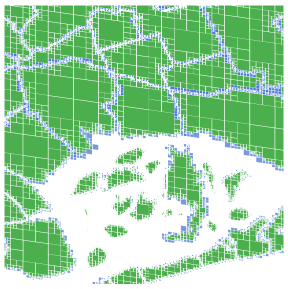


Figure 5: A super covering of neighborhoods in NYC's Jamaica Bay.

Figure 5 shows a super covering of neighborhoods in NYC's Jamaica Bay. Boundary (former covering) and interior cells are again marked in blue and green, respectively. Most of the area shown is covered by either interior cells or by no cells at all. Only

[1]Recall that a conflict between two cells exists if one cell contains the other. We do not consider duplicate cells as conflicting.

```
input:
 a list of coverings coverings // one per polygon
 a list of interior coverings interiors // one per polygon
output:
 // a list of (cell, polygon references)
 the super covering superCovering
procedure:
 for (covering in coverings) {
  for (cell in covering) {
   if (superCovering already contains cell) {
    add references of cell to existing cell
    continue
   }
   if (cell conflicts with existing cell in superCovering) {
    // cell is covered by existing cell or vice versa
    // resolve conflict
    $c_1$ = ascendant cell // may be cell or existing cell
    $c_2$ = descendant cell // may be cell or existing cell
    $d$ = difference of $c_1$ and $c_2$
    add references of $c_1$ to $d$ and $c_2$
    remove $c_1$ from superCovering // only required if the existing cell
      is the ascendant cell
    add $c_2$ and $d$ to superCovering
    continue
   }
   add {cell, {covering.polygonId, interior flag=false}} to superCovering
  }
 }
 // ... same code for interior coverings (with interior flag=true)
```

Listing 1: Build precision preserving super covering.

in the unlikely event that a query point hits a blue (boundary) cell, we may experience false positives (approximate approach) or we will need to enter the refinement phase (exact approach).

3.1.2 Data Structures. To store the super covering and enable efficient queries over it, we use two data structures: (i) a specialized radix tree (ACT) that indexes the cells of the super covering, and (ii) a lookup table that maintains the (variable-length) polygon references. Both data structures are designed for in-memory processing and are optimized for lookup performance.

Adaptive Cell Trie. ACT is a specialization of a textbook radix tree that indexes 64 bit cell ids. We call it adaptive for two reasons: (i) it indexes cells of adaptive sizes (to guarantee user-defined precision), and (ii) it can adapt to the expected point distribution. All adaptation is performed at build time. Once ACT is built, it is a static (immutable) data structure. We leave updates such as adding new polygons to an existing ACT for future work. However, we would like to point out that supporting updates is straightforward: In the build phase, cells of individual polygons are inserted one-by-one into ACT. The same procedure could be used to add new polygons at runtime, with appropriate synchronization between readers and writers. Code for removing polygons would follow the same logic, with the only difference being that we may want to (periodically) reorganize (i.e., compact) the lookup table.

We refer to the cell ids stored in the radix tree as *keys*. Each key denotes the path of a cell in the hierarchical grid. In the following, we first outline why a radix tree, in general, is a good choice for indexing quadtree cells, and we then explain how ACT differs from a general-purpose radix tree.

The main reasons why we choose a radix tree to index a super covering are (i) space efficiency and (ii) support for efficient prefix lookups. Compared to storing cell ids in a list, a radix tree avoids redundantly storing common prefixes, which reduces memory consumption. Prefix lookups, on the other hand, are required to find matching cells: The query point, which is a cell id at the most fine-grained grid level, is used to search for cell ids within the radix tree that share a common prefix (i.e., cover the query point). The runtime complexity of these lookups is in $O(k)$ with k being the key length, as opposed to the $O(\log n)$ of binary search that could be used on a sorted list. In other words, the number of node accesses in a radix tree is bounded by the maximum key length k_{max}, which is 60 when 30 quadtree levels are used (which is the case in our implementation). In practice, a lower k_{max} is often sufficient. For example, $k_{max} = 44$ allows for indexing cells up to level 22, which corresponds to a precision of less than 4 m (i.e., the distance between a point and a polygon in a false match is at most 4 m). A further advantage of the radix tree is that most queries can be answered using the upper levels of the tree: larger cells use fewer bits and are thus indexed closer to the root node. In the likely event that a query point hits a larger cell, we can complete the tree probe sooner.

We now discuss the design choices of ACT. The fanout f of the radix tree controls space consumption and lookup performance. A fanout of four means that we consume two bits at every tree level. With that configuration, our data structure matches the quadtree scheme (each node has four children, cf. Figure 6 for an example). While this would ease the implementation, it would require up to 30 nodes to be accessed per lookup. With a higher fanout, we can reduce this number. To maximize lookup performance, ACT uses a *default* fanout of 256 (= 8 bits). Thus, each level in ACT corresponds to four levels in the quadtree (each quadtree level is encoded with two bits). Let g be the cell level granularity of ACT (with $f = 256$, $g = 4$). While a fanout of 256 may result in sparsely occupied trie nodes, it allows for efficient lookups as it reduces the height of the trie to k_{max}/g. With $f = 256$, the maximum number of node accesses is $\lceil 60/\log_2(256) \rceil = 8$ for 30 quadtree levels.

Now we exploit a property of the hierarchical cells that we index: We *extend* their cell ids (keys) such that the key length matches the granularity of ACT. This process involves replacing a cell that we want to index with all its descendant cells at the next supported granularity level, and replicating the payload of the original cell to the smaller cells. In other words, if a cell does not match the tree granularity, we recursively split it into smaller cells that cover the same area. The following holds for indexed keys (cells):

$$level(cell) \bmod g = 0$$

Each cell c for which this equation does not hold is decomposed into a set of smaller cells C, with $|C| = 4^{g-(level(c) \bmod g)}$. This is possible since points are represented by cells at the most fine-grained grid level and use the maximum key length. Therefore, for a query point, it does not make a difference whether it matches with the originally inserted cell or with one of its descendant cells. This insight greatly simplifies the memory layout of a tree node and saves many CPU instructions: (i) we do not need to store the level with a cell, since all cells indexed in a tree node will have the same level, and (ii) a lookup in a node (an array) becomes a single offset access. Without this artificial key extension, we would need to perform multiple accesses per node to traverse all cell levels indexed in that node.

Figure 6 illustrates ACT indexing three polygons. While the example shows ACT with a fanout of four, by default we actually use a fanout of 256 to reduce the tree height. Every node thus consists of a fixed-sized array of 256 entries of 8 byte pointers. Entries that neither contain a child pointer nor a value point to a sentinel node indicating a false hit (no hit).

Values (i.e., polygon references) can be found in any level of the tree. This is because the indexed keys (64 bit cell ids in our case) typically use only a small fraction of the 64 bits with the remaining bits all set to zero. Larger cells that use fewer bits are indexed higher up in the tree, possibly even in the root node. In our example, polygon *a* is indexed by a cell in the upper level, while polygons *b* and *c* are indexed by cells in the lower level. Instead of storing values in separate nodes (e.g., adjacent to the tree nodes), we use combined pointer/value slots like in [25]. This design consumes less space and avoids an unnecessary indirection. Here, we exploit another property of the cell ids that we index: Cells in the super covering are disjoint, therefore a tree lookup will return at most one result. Due to this property, we never need to store a pointer and a value in an array entry at the same time. Using *pointer tagging*, we differentiate between pointers and values. We therefore refer to both pointers and values as *tagged entries*.

As stated before, each cell is associated with a set of polygon references. Thus, each value stored in the tree has to identify such a set. The canonical design would be to make each cell point to an entry in a lookup table that stores the references. However, at least in the case of largely disjoint polygons, cells mostly reference only one or two polygons. Therefore, to eliminate additional indirections, when there are no more than two polygon references, we store these references directly in the tree. A tagged entry can thus be:

- An 8 byte pointer to a child or the sentinel node (recall that a pointer to the sentinel node indicates a false hit)
- An inlined polygon reference (a 31 bit value)
- Two inlined polygon references (two 31 bit values)
- An offset (a 31 bit value) into a lookup table indicating that there are at least three polygon references

We use the two least significant bits of the 8 byte pointer to differentiate between these four possibilities. For an inlined polygon reference, we differentiate between a true hit and a candidate hit using the least significant bit of the 31 bit value. Thus, we can effectively only store 30 bit polygon ids (i.e., can index up to 2^{30} polygons).

We have experimented with path compression, but have found that storing common prefixes with inner and leaf nodes only barely reduces the number of nodes. Thus, the additional cache miss to access the prefix does not pay off. We therefore only use a common prefix at the root level.

We have also considered introducing adaptive node sizes, as proposed by the adaptive radix tree (ART) [25]. However, experiments have shown that introducing a second (compressed) node type with four children (Node4 in ART) (i) saves only a negligible amount of space for our workload and (ii) has a significant performance impact (due to the additional instructions and branch misses for dispatching between node types [25]). Also, lookups in compressed node types are more expensive.

Lookup Table. When a cell references more than two polygons, the tree contains an offset into a lookup table. Since cells often reference the same set of polygons, we only store *unique* polygon reference lists. The reference lists are split into two parts, a list with true hits and a list with candidate hits. Both lists contain polygon ids. The lookup table is encoded as a single 32 bit unsigned integer array. The offsets stored in the tree are simply offsets into that array. Each encoded entry contains the number of true hits followed by the true hits, the number of candidate hits, and the candidate hits.

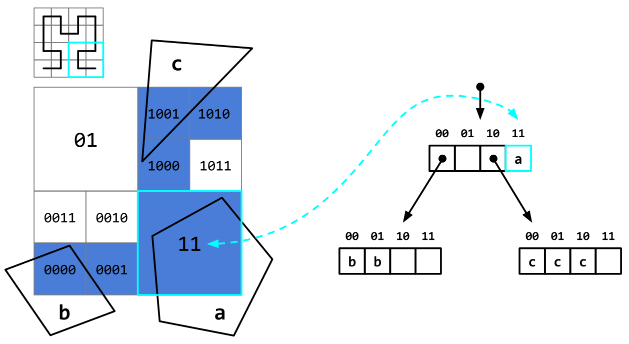


Figure 6: Adaptive Cell Trie indexing three polygons a, b, and c. Here, ACT uses two bits per level. In practice, we use up to eight bits (a fanout of 255) to reduce the tree height. Note that the figure only shows the cell rasterization for the part of the map that corresponds to the radix tree.

```
input:
 root node of ACT rootNode
 the cell id of the query point cellId
output:
 tagged entry taggedEntry
procedure:
 if (common prefix of rootNode does not match)
  return invalid entry
 level = 0
 currNode = rootNode
 bits = getBits(cellId, level++) // extract relevant bits
 // traverse the tree until we either hit the sentinel node or found a
     value
 while (taggedEntry = currNode.getEntry(bits) is a pointer) {
  if (taggedEntry points to the sentinel node)
   return false hit
  currNode = taggedEntry
  bits = getBits(cellId, level++)
 }
```

Listing 2: Probe Adaptive Cell Trie.

3.1.3 *Index Probing.* An ACT lookup returns, at most, one cell mapping to a set of polygon references. Listing 2 shows the probe algorithm. While traversing the radix tree does not involve any key comparison, a comparison is performed to check whether the returned tagged entry contains a payload. For that, we need to differentiate between (i) one polygon reference, (ii) two polygon references, and (iii) an offset. In the first case, we check whether the polygon reference is invalid, which indicates a false hit. Otherwise, we extract the interior flag (the least significant bit of the 31 bit payload) and the polygon id and return the reference. In the second case, we extract and return both references. Only in the third case, we need to access the lookup table to retrieve the polygon references.

3.2 Approximate Join with Precision Bound

The complete point-polygon join algorithm is shown in Listing 3. It is essentially an index nested loop join, using our novel ACT

```
input:
 points points // lat/lng coordinates and cell ids
 polygons polygons // lat/lng coordinates of vertices
 root node rootNode
 lookup table lookupTable
output:
 list of join pairs pairs // point/polygon pairs
procedure:
 for (point in points) {
  taggedEntry = probeAdaptiveCellTrie(rootNode, point.cellId) //
      cf., Listing 2
  if (taggedEntry is invalid)
   continue
  references = getPolygonReferences(lookupTable, taggedEntry) //
      returns a list of polygon references
  for (reference in references) {
   polygonId = reference.polygonId
   polygon = polygons[polygonId]
   if (reference is true hit) {
    add {point, polygon} to pairs
   } else { // candidate hit
#ifdef __APPROX
    // treat candidate hit as true hit
    add {point, polygon} to pairs
#else
    // EXACT: enter refinement phase
    if (polygonCoversPoint(polygon, point)) // PIP test
     add {point, polygon} to pairs
#endif
 }}}
```

Listing 3: The join algorithm.

index that makes the point-cell containment tests very efficient. For a given point, we retrieve the cell that contains it (if such a cell exists) and go over all references of this cell. When approximate results are sufficient, we omit the expensive refinement phase, simply treat all points contained in boundary cells as (approximate) hits, and immediately output the join pairs. In doing so, we introduce false positives. However, the distance of false positives from the polygon is bounded by the diagonal of the largest boundary cell: Any point contained in that cell has at most a distance of $\sqrt{2} * \epsilon$ (with ϵ being the side length of the cell) to the polygon. In order to control this distance, our approximate algorithm exposes a precision bound as a parameter to the user. Based on this bound, we compute the minimum cell level for boundary cells. For example, to guarantee a 4 m precision, the largest boundary cell can at most have a diagonal of 4 m, which corresponds to a minimum cell level of 22 in our implementation (i.e., cell level 21 would be too coarse-grained). We replace all boundary cells in the super covering with their descendant cells at the required level. For each of these descendant cells, we determine whether they intersect, are fully contained in, or do not intersect polygons at all, and update ACT accordingly: We remove the original cell c_o from ACT and insert *only* those descendant cells that intersect or are fully contained in polygons. The new cells may reuse the lookup table entry of c_o or create their own in the event that they only map to a subset of c_o's polygons.

Note that [39] makes use of a similar distance-based precision bound, however, uses a single-resolution grid. When it is not possible to maintain a sufficiently fine-grained index within a certain memory budget, the user can fall back on our accurate approach, in which we train the index with historical data points.

3.3 Accurate Join

When applications require accurate results, or when we cannot build an index that satisfies a user-defined precision without exceeding a memory budget, we use an approach that may enter the expensive refinement phase (cf. Listing 2). To minimize the number of (expensive) PIP tests, we increase the precision of the index by adapting it to the expected point distribution. Since we make use of true hit filtering, a finer-grained index allows us to identify more join partners during the filter phase.

3.3.1 Index Training. To minimize the likelihood of PIP tests, we train the index to adapt to the expected distribution of query points. We train ACT with historical data points (e.g., from a previous year) which has the effect that popular areas that expect more hits are approximated using a more fine-grained grid than less popular areas. This training process replaces *expensive* cells with up to four of their child cells. We define expensive cells as cells that map to polygon reference sets with at least one candidate hit. When we hit such a cell during the join, we need to perform expensive PIP tests.

Specifically, the training works as follows: When a training point hits an expensive cell, for each of its four child cells we check whether they intersect, are fully contained in, or do not intersect the referenced polygons at all, and update ACT accordingly. The cell replacement procedure is the same as for the approximate algorithm (i.e., remove original cell, insert descendant cells, and update lookup table, cf. Section 3.2) with the only difference being that we always replace an expensive cell with its direct children one level below. We do not replace a cell with even smaller cells to be more robust against outliers. In practice, we would stop refining the index once a user-defined memory budget is exhausted. In this work, we focus on training the index in a dedicated training phase. Training the index at runtime would introduce additional concurrency and buffer management issues that we leave for future work. We show the effect of training the index in Section 4.2.

3.4 Implementation Details

Join Predicate. Our current implementation follows the semantics of the `ST_Covers` join predicate (cf. PostGIS [30]). `ST_Covers` evaluates whether one geospatial object (e.g., a polygon) covers another (e.g., a point).

Individual Polygon Coverings. We compute the individual polygon coverings using the S2 library. Note that our approach does not depend on S2 and, in fact, works with any other quadtree-based hierarchical grid in which each (implicit) quadtree node [16] corresponds to a geographical area (space partitioning). For our approach to work, each quadtree node needs to be *uniquely identifiable* with a bit sequence that represents the path to the given node starting from the root. Thereby, any (consistent) enumeration scheme (e.g., the Hilbert space-filling curve used by S2 or the Z curve used by Roth [31]) of the four quadrants is valid. To store these encoded node identifiers in a trie, we require the identifiers of child nodes to share a common prefix with their parent node.

Face Nodes. Since our implementation uses S2, which projects points on Earth onto a surrounding cube, we need to maintain up to six radix trees (one for each face). Using the first three bits of the query cell id, we select the appropriate radix tree.

Index Probing. The probe (filter) phase is the performance-critical part of our approach. We therefore parallelize this phase to accelerate lookups in the radix tree. Individual processing threads fetch batches of 16 tuples at a time and synchronize using an atomic counter.

PIP Test. In the refinement phase, we use S2's PIP test, which implements the ray tracing algorithm (cf. [29] for performance numbers).

4 EXPERIMENTAL EVALUATION

In this section, we present a thorough experimental analysis of our point-polygon join algorithms. We use taxi data from NYC, which we join with different polygonal regions of NYC, such as neighborhoods. We also experiment with geo-tagged Twitter data from different cities. Besides these (skewed) real-world point datasets, we experiment with (uniform) synthetic point data. We focus our experiments on the probe phase of the join (probing points against a pre-built polygon index). For completeness, we also report build times.

Our evaluation is structured into three parts: First, we evaluate the performance and space consumption of our approximate algorithm with different data structures, including ACT, a B-tree, and a sorted vector. We demonstrate that for a city like NYC (with its 289 neighborhoods), an approximate index with very high precision (<4 m precision bound) easily fits into the main memory of a single machine and, in the case of ACT, allows for very high probe performance (>50 M points/s per CPU core). We think that this is a good fit with the city-centric model of mobility companies (e.g., Uber, DriveNow [13]). We show that ACT outperforms other physical representations by a large margin, while being more space-efficient in many cases. Second, we evaluate our accurate algorithm and show that it benefits greatly from true hit filtering. We compare it against other filter and refine approaches, including an R-tree on the polygons' MBRs, a geospatial index by Google, and PostgreSQL (PostGIS). We demonstrate that the high precision of our index can be further improved by training it with historical data points. Third, we show that both our algorithms are competitive with state-of-the-art GPU approaches.

Infrastructure. We use a server-class machine that is equipped with two 14-core Intel Xeon E5-2680 v4 CPUs and 256 GiB DDR4 RAM. All CPU-based approaches are implemented in C++ and compiled with GCC version 5.4.0 with `O3` and `march=core-avx2` flags. We conduct the experiments on a single socket to eliminate NUMA effects. For the comparison against the GPU join algorithms, we use these Amazon Web Services (AWS) instances [5]:

c5.4xlarge	16 vCPUs, USD 0.68/hour
g3s.xlarge	NVIDIA Tesla M60 GPU, USD 0.75/hour

Datasets and Queries. We use 1.23 B points (pick-up locations) from the NYC yellow taxi dataset (years 2009 to 2016), which is publicly available in CSV format [38]. For each point, we load its lat/lng coordinate and convert it to an `S2Point` [33] (which represents a point on the unit sphere as a 3D vector of doubles) and to an `S2CellId` (an 8 byte value, cf. Section 2) prior to performing any experiments. We maintain one `std::vector` of `S2Points` and another one storing the corresponding cell ids. We join these points against the polygon datasets summarized in Table 1 (top). All three polygon datasets cover approximately the same area. While there are only five boroughs, their polygons are significantly more complex.

In addition, we use geo-tagged tweets collected from Twitter's live public feed over a period of five years. From these, originally over 2.29 B tweets spread across the entire US, we extract four point datasets based on the MBRs of NYC, Boston (BOS), Los Angeles (LA), and San Francisco (SF), consisting of 83.1 M, 13.6 M, 60.6 M, and 9.57 M points, respectively. We join these points against the corresponding neighborhood polygons: NYC (289), BOS (42), LA (160), and SF (117). Since we extract the points using the MBR of the entire polygon dataset and not the individual neighborhood polygons, there are points that do not join with any polygon.

We also generate synthetic point datasets, uniformly distributed within the MBR of the respective polygon dataset.

We focus our experimental evaluation on the probe phase and simply count the number of points per polygon instead of materializing the join result. To avoid any contention in the multi-threaded experiments, we maintain thread-local counters that we aggregate in the last step. Since we are focusing on the case of static polygons, the reported throughput times reflect the time to compute the counts using an *existing* (pre-built) polygon index. We report the time it takes to build the polygon index separately. However, we would like to point out that we did not optimize the build phase.

Polygon Approximations. Our default configuration for computing the individual polygon coverings is as follows: max covering cells = 128, max covering level = 30, max interior cells = 256, and max interior level = 20.

4.1 Approximate Join

We first analyze the performance and space consumption of our approximate algorithm. In all of the following experiments, we first build super coverings (sets of cell/value pairs, cf. Section 3) and then index them with different data structures.

Super Covering Construction. Table 1 shows different metrics of the super coverings for the three polygon datasets with 60 m, 15 m, and 4 m precision. With each cell occupying 64 bits, the largest super covering (census 4 m, 39.8 M cells) amounts to 304 MiB of raw key data and another 304 MiB for the values (64 bit tagged entries, cf. Section 3). Given that most cells reference fewer than three polygons, most polygon references are inlined, which keeps the lookup table small. While the computation of the individual coverings is parallelized over the number of polygons, the construction of the super covering is performed serially.

Data Structures. We essentially need to map cell ids (64 bit integers) to tagged entries (64 bit values). A tagged entry either contains up to two polygon references or an offset into a lookup table. The lookup table is the same among all data structures that we evaluate. The data structure needs to support prefix lookups: given a 64 bit lookup key (the cell id of a query point), find the cell in the super covering (recall that it only contains non-overlapping cells) that shares a common prefix with the lookup key (if such a cell exists). We analyze ACT with three different fanouts: 2, 4, and 8 bits per radix level, which corresponds to 1, 2, and 4 quadtree levels, respectively. Recall that one quadtree level is encoded with two bits. We therefore refer to these three variants as ACT1, ACT2, and ACT4. As competitors we use a B-tree implementation by Google [11] (GBT) and a binary search on a sorted vector implemented with `std::lower_bound` (LB). For GBT, we use a (target) node size of 256 bytes, which turned out to be the most query-efficient configuration. The vector stores pairs of cell ids and tagged entries. We have also experimented with the STX B+-tree [36] but do not include it in this section as its lookup performance is very similar to that of GBT.

The performance of our approximate algorithm is dominated by the costs of the ACT node accesses and the aggregation (count).

Table 1: Metrics of the NYC polygon datasets and of three super coverings with various precisions.

polygons (# polygons / avg. # vertices)	boroughs (5 / 662)			neighborhoods (289 / 29.6)			census (39,184 / 12.5)		
precision [m]	60	15	4	60	15	4	60	15	4
# cells [M]	0.09	1.32	20.9	0.16	0.98	14.0	8.50	8.97	39.8
lookup table [MiB]	0.00	0.00	0.00	0.01	0.01	0.01	1.33	1.33	1.41
build individual coverings [s]	0.11	0.98	16.0	0.07	0.19	1.54	0.96	1.01	3.08
build super covering [s]	0.10	0.94	15.2	0.17	0.81	10.5	11.6	11.8	37.7

Table 2: Metrics of the different data structures (4 m precision).

super cov.	boroughs (20.9 M cells)					neighborhoods (14.0 M cells)					census (39.8 M cells)				
index	ACT1	ACT2	ACT4	GBT	LB	ACT1	ACT2	ACT4	GBT	LB	ACT1	ACT2	ACT4	GBT	LB
size [MiB]	328	198	173	359	319	224	138	143	240	214	624	421	1234	684	608
build [s]	2.11	1.46	1.06	1.39	-	1.36	0.98	0.69	0.85	-	4.00	3.11	2.80	2.85	-

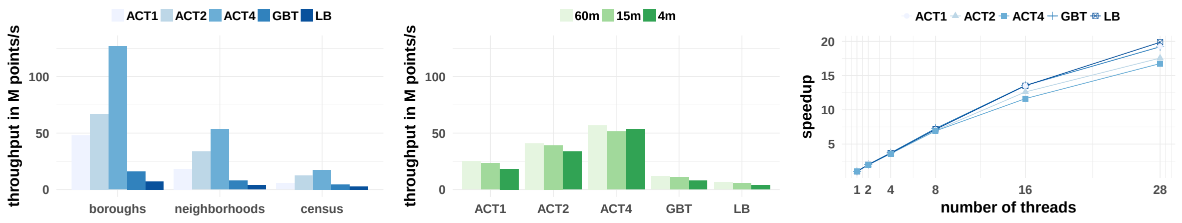


Figure 7: Throughput and scalability of our approximate algorithm (taxi dataset). Left: Single-threaded execution with different data structures (4 m precision). Middle: Single-threaded execution with different precisions and data structures (neighborhood polygons). Right: Multi-threaded execution (neighborhood polygons, 4 m precision).

To better understand the results, we therefore first analyze the space consumption of ACT and compare it with GBT and the sorted vector. Table 2 shows size and build time (single threaded) of the different data structures on the super coverings introduced above (4 m precision only). In many cases, ACT consumes less space than the sorted vector (LB). Due to the high density of the cell ids, ACT is more space-efficient with higher fanouts, except for census where ACT4 consumes the most space: Like for all datasets, ACT4 has fewer (but larger) nodes than ACT1 and ACT2. However, in this case, its nodes are very sparsely populated compared to those of ACT1 and ACT2. The reason is that ACT4's nodes cover too much space for the relatively small census cells. To mitigate the size impact of sparse ACT nodes, one can represent them with a more compact data structure [4]. All 4 m indexes exceed the 35 MiB L3 cache of our evaluation machine. Note that there is no additional build time for LB, since the super covering contains cell id/tagged entry pairs already sorted by cell id.

Single-Threaded Throughput. For this experiment, we compute a super covering with a 4 m precision bound on the three NYC polygon datasets and store it in the different data structures introduced above. We then join the full taxi dataset (all 1.23 B points) against each of these indexes and report the throughput in M points/s (cf. Figure 7 (left)).

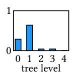
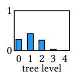
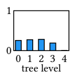
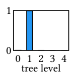
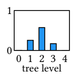
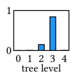
ACT clearly dominates the B-tree and the binary search on the sorted vector, especially in its highest fanout configuration (ACT4). A higher fanout means that we consume more bits of the lookup key per tree level and thus require fewer node accesses (i.e., need to traverse fewer levels) to find a key (an indexed cell). With ACT4 for example, we consume 8 bits per tree level and thus need at most 64/8 = 8 node accesses. Since we reduce the tree height further by storing a common prefix at the root level (cf. Section 3), ACT requires even fewer node accesses (e.g., at most five with 4 m precision).

Table 3: Speedups of lookups in smaller (more coarse-grained) over larger (more fine-grained) polygon datasets for different data structures (b = boroughs, n = neighborhoods, c = census).

	b over n	b over c	n over c
ACT1	**2.63×**	**8.63×**	**3.28×**
ACT2	2.00×	5.33×	2.66×
ACT4	2.36×	7.29×	3.08×
GBT	2.05×	3.51×	1.71×
LB	1.83×	2.63×	1.44×

Another insight is that ACT benefits the most from the larger (coarser-grained) cells in the smaller polygon datasets as shown in Table 3. Going from the most fine-grained census dataset (39,184 polygons) to the most coarse-grained boroughs dataset (5 polygons), GBT's lookup performance improves by 3.51×, while ACT1's increases by 8.63×. The reason for ACT's large gain is that larger cells are indexed higher up in the radix tree and are thus found sooner. GBT, in contrast, does not benefit from these larger cells, which might as well be stored in the leaf nodes of the B-tree. GBT's performance gain comes from the smaller number of cells used for indexing the boroughs dataset and the resulting smaller B-tree (i.e., fewer branch and cache misses per point).

Table 4: Distribution of the tree traversal depth (ACT4 with 4 m precision).

points	boroughs	neighborhoods	census
uniform			
taxi			

Likewise, the binary search on the sorted vector (LB) is only affected by the number of cells and not their granularity.

Different Precisions. Next, we vary the precision of the indexed super covering. We perform this experiment using the medium size neighborhoods dataset. Figure 7 (middle) shows the throughput numbers for the different data structures. While GBT's and LB's performance decreases by 33.4% and 39.4%, respectively from 60 m to 4 m, ACT4's performance is hardly affected (-5.73%) by the larger number of cells of the more precise super covering. Compared to the 60 m covering, the more precise coverings contain a larger number of small cells (in the boundary areas of the polygons). Query points are unlikely to hit these cells in contrast to the large (more coarse-grained) cells, which are indexed in the upper (cached) ACT nodes (due to their shorter cell ids). ACT1 and ACT2 are more affected by the precision increase (-27.8% and -17.9%, respectively). The reason is that the added small cells have a stronger effect on the depths of these trees. While the average node depth for ACT4 only increases from 2.83 to 2.97 (+4.95%) from 60 m to 4 m respectively, the same metric increases from 10.8 to 14.6 (+35.2%) for ACT1. Although—as already stated above—the new small cells are unlikely to be hit, they still cause a performance hit for lower fanouts.

ACT4's throughput is similar for 15 m and 4 m (-4.15%) because its structure is identical for both precisions. In both cases, it has 70,786 nodes occupying 143 MiB. The only difference is the nodes' structure: Due to the more fine-grained cell approximation, the average node occupancy (measured in terms of occupied slots) of ACT4 at tree level 3 decreases from 88.2% (60 m) to 85.2% (4 m). The occupancies of all other levels are the same. This lower occupancy for 4 m saves some aggregations (for updating the polygon hit counts), causing a slightly higher performance.

In summary, the impact of precision on query performance is less significant for ACT than for the other data structures.

Multi-Threaded Throughput. In this experiment, we study the lookup performance of the different data structures with an increasing number of threads on the neighborhoods dataset with a 4 m precision bound. We use up to 28 threads, which matches the number of hyperthreads of a single NUMA node of our evaluation machine. Figure 7 (right) shows the speedups over single-threaded execution. Up to 8 threads, all index structures scale almost linearly (speedup of around 7× in all cases). This is what we would expect for immutable data structures.

The fact that an oversubscription of cores (hyperthreading) has a positive performance impact suggests that the lookup is bound by memory access latencies (having more threads than physical cores can hide these latencies).

Synthetic Points. To show the general applicability of our approach, we also experiment with synthetic point data. We generate 100 M points uniformly distributed within the MBR of the respective (NYC) polygon dataset. Table 4 shows the probability distribution of the number of search steps during the tree traversal for the synthetic and the taxi point dataset. As expected, the distribution for the uniform data is skewed towards the root. That is because the larger cells (which are more likely to be hit) are indexed closer to the root. The distribution for the taxi data depends on the polygon dataset. For boroughs, most traversals end at tree level 1, while for census, points mostly hit small cells indexed in tree level 3.

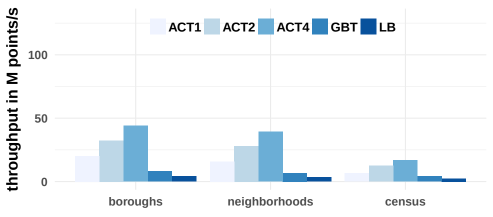


Figure 8: Single-threaded throughput of our approximate algorithm (4 m precision) with uniform point data.

Figure 8 shows the single-threaded throughput for the different data structures with the uniform point data. ACT achieves the highest throughput, with ACT4 again being the most query-efficient configuration. The absolute numbers, however, are lower than for the (real-world) taxi data: ACT4's throughput decreases by 65.2%, 26.8%, and 3.11% for boroughs, neighborhoods, and census, respectively.

The reason for this slowdown is simple: The synthetic point data is uniformly distributed, which leads to more branch and cache misses (cf. Table 5 for performance counters on neighborhoods). In contrast, the real-world taxi data is highly clustered with the majority of points located in Manhattan (>90%) and around the airports. For boroughs (not shown in Table 5), ACT4 endures 0.79 and 0.01 branch misses per point for the synthetic and the taxi points, respectively. This is the main cause of the 65.2% performance drop mentioned above.

Twitter Data. Next, we analyze the performance of our approach on the four Twitter datasets and the corresponding neighborhood polygons (cf. Figure 9). The numbers are similar across the different cities, with the highest throughput achieved for BOS with its only 42 neighborhood polygons. Next comes SF followed by LA and NYC, for which the throughput is very close to what we obtained with the taxi data (cf. Figure 7 (left)). In fact, with a 4 m precision, ACT4 achieves a single-threaded throughput of 52.1 M points/s, which is almost the same as the 53.6 M points/s on the taxi data. Similarly to the taxi points, the tweets are clustered, with certain areas having more tweeting activity than others. In contrast, with uniform point data, ACT4 only achieved 39.3 M points/s. This confirms that our approach benefits from the skewed distribution of real-world data. For all four cities, the numbers are (again) hardly affected by the precision.

4.2 Accurate Join

We now evaluate our accurate algorithm, which eliminates false positives in an additional refinement phase. We demonstrate that our index benefits significantly from true hit filtering and that index training with historical data can further improve its effect.

Competitors. We compare against the boost R-tree (1.6.0) [8] on the polygons' MBRs (RT), Google's S2ShapeIndex [34] (SI), and PostgreSQL 9.6.1 (PostGIS 2.3.1) [30] with a GiST index on

Table 5: Performance counters per point (neighborhoods, 4 m precision).

points	uniform					taxi				
index	ACT1	ACT2	ACT4	GBT	LB	ACT1	ACT2	ACT4	GBT	LB
cycles	154	99.8	71.3	415	569	172	93.8	56.4	416	817
instructions	214	121	82.4	486	927	202	121	81.3	393	564
branch misses	1.06	1.04	0.88	5.32	8.38	0.96	0.83	0.48	7.06	10.8
cache misses	0.29	0.23	0.18	0.70	1.89	0.22	0.17	0.15	0.29	0.37

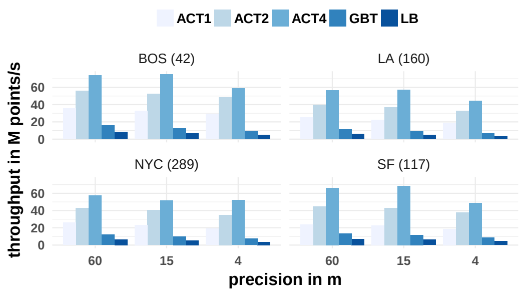


Figure 9: Single-threaded throughput of our approximate algorithm (Twitter datasets, polygon counts in brackets).

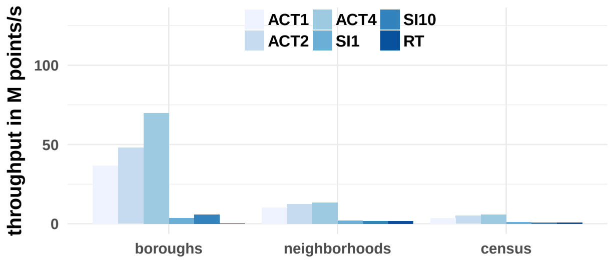


Figure 10: Single-threaded throughput of our accurate algorithm (with different ACT fanouts) compared to S2ShapeIndex (with 1 and 10 edges per cell) and the R-tree.

polygons (PG). Our algorithm and the R-tree both use the same PIP test implementation (cf. Section 3.4). SI also uses that implementation, however, restricts the test to a subset of edges of the polygon in question. This is achieved by using a hierarchical grid approximation of polygons, and internally mapping grid cells (64 bit S2CellIds) to polygon edges using a B-tree. This hierarchical grid approximation is much more coarse-grained than our super covering, given its higher focus on build time than on query performance (compared to our approach). SI allows the maximum number of edges per cell to be configured, essentially controlling the granularity of the employed grid approximation. We evaluate SI with its default configuration of 10 edges (SI10) and 1 edge per cell (SI1). Note that SI1 is the most fine-grained configuration possible. SI also employs true hit filtering (cf. Section 3) to avoid PIP tests, but in a much less effective way than ours (due to its coarser-grained grid). Furthermore, SI does not offer an approximate version. For the R-tree, we use the splitting strategy `rstar` with at most 8 elements per node which performs best in all workloads.

Taxi Data. For this experiment, we compute coarse-grained super coverings that do not guarantee a certain precision, and instead fall back on a refinement phase for candidate hits. Here, the resolution of a super covering is determined by our default configuration for computing individual polygon coverings introduced earlier (cf. Section 4). Thus, these super coverings consist of much fewer cells than those guaranteeing a certain precision. For example, the approximation for the neighborhoods dataset now only consists of 98,687 cells (ACT4 size: 25.9 MiB) compared to the 13.2 M cells (ACT4 size: 143 MiB) needed to guarantee a 4 m precision. For this dataset, SI1, SI10, and RT consume 1.20 MiB, 0.23 MiB, and 27.9 KiB, respectively.

Figure 10 shows the single-threaded throughputs for the accurate join. ACT4 achieves the highest performance for all three datasets. For the medium size neighborhoods dataset, it outperforms SI1 by 6.96×, followed by SI10, which is only 7.41% slower than SI1. For census, ACT4 still outperforms SI1 by 5.79×. RT has the lowest numbers with 0.21, 1.77, and 0.79 M points/s for boroughs, neighborhoods, and census, respectively. The reason for its slow performance for boroughs is as follows: The complexity of each PIP test (ray-tracing algorithm) is linear with the size (number of edges) of the polygon. Since the boroughs are complex polygons with many edges, the PIP tests in the refinement phase are very expensive. Here, our algorithm shines since it can identify most join partners in the filter phase and only enters the refinement phase for 0.1% of the points. As a point of reference, PG achieves 0.39, 1.09, and 0.69 M points/s for boroughs, neighborhoods, and census, respectively (because we use all hyperthreads on our evaluation machine, PG's numbers are not directly comparable and are excluded from the plot). Similar to RT, PG is affected by the complex boroughs polygons.

Index Training. As readers may have noticed, there is a large performance gap between our approximate and our accurate algorithm. For example, ACT4 (accurate) is 75.3% slower than its approximate counterpart (with 4 m precision) on the taxi data/neighborhoods join. The reason is the expensive PIP tests needed to compute an accurate result.

We now show how to narrow this performance gap. The idea is to reduce the likelihood for PIP tests by training the index with historical data points (cf. Section 3.3.1). In other words, we increase the precision of the index by making it more fine-grained in areas where we expect more points. One effect this has is that the size of the area covered by (expensive) boundary cells will decrease. We train the index with taxi points sampled from the year of 2009 and only use the points from 2010 to 2016 for the join. Table 6 shows the performance impact. With 100 K training points, ACT4's performance improves by 1.56× for neighborhoods and increases further to 2.18× with 1 M points (due to a 84.0% reduction in the number of PIP tests). The size of ACT4 only increases from 25.9 MiB (untrained) to 28.0, 34.8, and 44.3 MiB when trained with 100 K, 500 K, and 1 M historical data points, respectively. In absolute terms, ACT4 trained with 1 M points achieves a throughput of 29.1 M points/s for neighborhoods and

Table 6: Speedups of single-threaded lookups when training ACT4 with an increasing number of historical data points (over untrained ACT4).

no. of train. points	boroughs	neighborhoods	census
100 K	1.25×	1.56×	1.16×
500 K	1.40×	2.00×	1.40×
1 M	1.44×	2.18×	1.53×

Table 7: Effect of training the index with 1 M historical data points (STH = solely true hits).

metric	boroughs	neighborhoods	census
STH (%)	99.9 → 99.9	87.2 → 97.7	72.2 → 88.7

thus narrows the performance gap to its approximate counterpart (with 4 m precision) from 75.3% to 45.7% while consuming 68.9% less space. This shows that a trained accurate index is a good alternative to our approximate indexes when main memory is sparse. Table 7 shows the effect of true hit filtering when training the index with 1 M training points. The metric `solely true hits` (STH) indicates the percentage of points that skipped the expensive refinement phase, which is clearly above 70% in all cases (even without training). Training the index significantly improves STH for neighborhoods and census.

4.3 Comparison with GPU Algorithms

Finally, we compare our approximate and accurate (untrained ACT) algorithms against state-of-the-art GPU counterparts [39]. The GPU approaches leverage the graphics rendering pipeline, and in particular the rasterization operation, which converts a polygon into a collection of (equi-sized) pixels. Similar to our approach, the GPU join also comes in two variants: Bounded Raster Join (BRJ), which guarantees a user-defined precision by appropriately scaling the rendering resolution, and Accurate Raster Join (ARJ), which performs PIP tests for points falling on the pixels forming the boundaries of the polygons. To enable a fair comparison, we do not consider any preprocessing times on the polygons (such as triangulation time). Note that the preprocessing time for the GPU join is minimal. In fact, it is designed for computing the join on-the-fly without a priori knowledge of the polygonal regions.

We now compare the throughput of both approaches on two similarly priced AWS machines (cf. Infrastructure in Section 4). Figure 11 shows the results of joining 612 M taxi rides with the NYC polygon datasets. While our approximate algorithm is again hardly affected by the precision (15 m vs. 4 m), BRJ takes a significant performance drop. The reason for BRJ's slowdown is simple: Once the required resolution is higher than what is natively supported by the GPU, it needs to split the scene and perform more rendering passes. This is essentially related to the fact that BRJ relies on a uniform grid. On the contrary, BRJ is barely affected by the polygon datasets, while our approximate algorithm is. The reason is again related to the granularity of the grid: With the more fine-grained census dataset, we need to traverse more tree nodes (as the cells that approximate the polygons are smaller), while the rendering resolution in BRJ depends only on the size of the bounding box of the polygon dataset and the precision. With

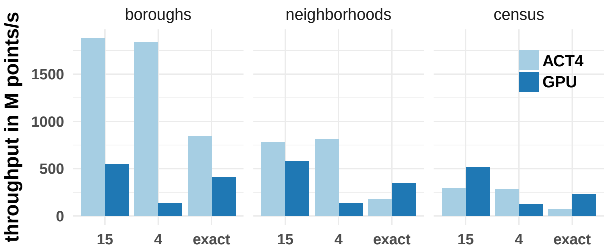


Figure 11: Throughput of ACT4 (16 threads) compared to the two GPU algorithms on AWS (GPU = Bounded Raster Join for 15 m and 4 m and Accurate Raster Join for exact).

exact results, our approach outperforms ARJ for boroughs, while ARJ takes the crown for neighborhoods and census.

5 RELATED WORK

Prior Publications. In [24], we describe a novel approach to reduce control flow divergence on AVX-512 platforms to further increase ACT's lookup performance. Note that [24] is based on an earlier (4-page) version of this work [22].

Spatial Join Techniques. The point-polygon join is one of the core operations in spatial databases, and, a large body of related work on algorithmic techniques [19] is available accordingly.

Naturally, we are not the first to index polygons using raster approximations. Early on, Orenstein [27] proposed decomposing single polygons into multiple cells. Later, Brinkhoff et al. [9] proposed true hit filtering in the form of maximum enclosed rectangles and circles, allowing the refinement phase to be skipped in many cases. Zimbrao et al. [49] followed up on this approach by using raster approximations in the form of uniform grids, thereby improving selectivity. Kothuri et al. [20] recursively divide the MBR of a polygon into four cells until a certain granularity is reached, identify interior cells, and index them in an R-tree to skip refinement checks. The primary goal was to minimize I/O, an important performance factor for disk-based systems. In contrast to these early works on true hit filtering and also to the recent proposal by Tzirita Zacharatou et al. [39], we use a quadtree-based (multi-resolution) grid that can be very coarse-grained in interior and very fine-grained in boundary areas.

Research has, however, also been performed on true hit filtering with quadtree-based rasterizations, including work in Oracle Spatial [21] and Microsoft SQL Server [15]. In both of these works, *individual* polygons are approximated using a set of multi-resolution grid cells. These grid cells are enumerated using one-dimensional cell identifiers and stored in a B-tree. In contrast, we *holistically* approximate and index an entire set of polygons and store these (in our case duplicate-free) cell identifiers in a novel radix tree (ACT), which is more query-efficient than a B-tree. Additionally, these existing approaches neither offer an approximate mode nor allow the accurate index to be trained with historical data points to improve query performance.

To decrease the probability of false matches, [35] improves the precision of MBRs by clipping away empty space that is concentrated around the MBR corners. In contrast to our work, [35] uses the classical filter and refine evaluation strategy.

Related to our approximate algorithm is work by Azevedo et al. [7] that provides precision estimates for approximate polygon-polygon joins using a less space-efficient single-resolution grid. Tzirita Zacharatou et al. [39] propose a similar precision bound to ours but also use a single-resolution grid (cf. Section 4.3 for a comparison).

The PH-tree [47] is another example of a trie data structure that indexes multi-dimensional data. In contrast to ACT, it only indexes points, not higher-level grid cells. Winter et al. [43] propose a query-efficient storage layout for point data that automatically adapts to polygonal queries. Along the same lines, Vorona et al. [42] train a model to approximately answer spatial aggregation queries.

Systems. Several database systems support geospatial joins. PostGIS [30], a geospatial extension to PostgreSQL [1], uses an R-tree implemented on top of GiST [18] for indexing geospatial objects. In recent years, various spatial data management systems based on Hadoop [3, 14] and Spark [26, 37, 44, 46] have emerged. [28] provides a comprehensive analysis of these modern spatial analytics systems by a thorough experimental evaluation. In contrast to our work, most of these systems rely on offline partitioning of the data points.

Modern Hardware. Most work on using modern hardware for geospatial joins focuses on GPU offloading [2, 12, 39, 45, 48] while [10] proposes a GPU-accelerated end-to-end spatial system.

6 CONCLUSIONS

We have presented two point-polygon join algorithms that use a multi-resolution grid indexed in a query-efficient radix tree. We have transformed a traditionally compute-intensive problem into a memory-intensive one. We have shown that it is possible to refine the index up to a user-defined precision and identify all join partners in the filter phase. We have demonstrated that the accurate version of our algorithm can adapt to the expected point distribution. We have also shown that our approach outperforms existing CPU-based joins by up to two orders of magnitude and can compete with dedicated GPU implementations.

ACKNOWLEDGMENTS

This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) grants `01IS12057` (FASTDATA) and `01IS18025A` (BBDC II). It is further part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government. Harish Doraiswamy was supported in part by the Moore-Sloan Data Science Environment at NYU, and NSF awards `CNS-1229185`, `CCF-1533564`, `CNS-1544753`, `CNS-1730396`, and `CNS-1828576`.

REFERENCES

[1] PostgreSQL. http://www.postgresql.org/.
[2] D. Aghajarian and S. K. Prasad. A spatial join algorithm based on a non-uniform grid technique over gpgpu. In *Proc. of SIGSPATIAL*, pages 56:1–56:4, 2017.
[3] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz. Hadoop-GIS: A high performance spatial data warehousing system over MapReduce. *PVLDB*, 6(11):1009–1020, 2013.
[4] C. Anneser, A. Kipf, H. Lang, T. Neumann, and A. Kemper. The case for hybrid succinct data structures. In *Proc. of EDBT*, 2020.
[5] AWS instance types. https://aws.amazon.com/ec2/instance-types/.
[6] L. G. Azevedo, R. H. Güting, R. B. Rodrigues, G. Zimbrão, and J. M. de Souza. Filtering with raster signatures. In *Proc. of ACM-GIS*, pages 187–194, 2006.
[7] L. G. Azevedo, G. Zimbrão, and J. M. de Souza. Approximate query processing in spatial databases using raster signatures. In *Proc. of GEOINFO*, pages 53–72, 2006.
[8] *boost::geometry::index::rtree - 1.60.0.* https://www.boost.org/doc/libs/1_60_0/libs/geometry/doc/html/geometry/reference/spatial_indexes/boost__geometry__index__rtree.html.
[9] T. Brinkhoff, H. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proc. of SIGMOD*, pages 197–208, 1994.
[10] H. Chavan, R. Alghamdi, and M. F. Mokbel. Towards a GPU accelerated spatial computing framework. In *ICDE Workshops*, pages 135–142, 2016.
[11] Google C++ B-tree. https://code.google.com/archive/p/cpp-btree/.
[12] H. Doraiswamy, E. Tzirita Zacharatou, F. Miranda, M. Lage, A. Ailamaki, C. T. Silva, and J. Freire. Interactive visual exploration of spatio-temporal urban data sets using urbane. In *Proc. of SIGMOD*, pages 1693–1696, 2018.
[13] DriveNow. https://www.drive-now.com/de/en.
[14] A. Eldawy. SpatialHadoop: Towards flexible and scalable spatial processing using MapReduce. In *SIGMOD, PhD Symposium*, pages 46–50, 2014.
[15] Y. Fang, M. Friedman, G. Nair, M. Rys, and A. Schmid. Spatial indexing in microsoft SQL server 2008. In *Proc. of SIGMOD*, pages 1207–1216, 2008.
[16] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, 1982.
[17] E. Haines. Point in polygon strategies. *Graphics gems IV*, 994:24–26, 1994.
[18] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. of VLDB*, pages 562–573, 1995.
[19] E. H. Jacox and H. Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1):7, 2007.
[20] K. V. R. Kanth and S. Ravada. Efficient processing of large spatial queries using interior approximations. In *Proc. of SSTD*, pages 404–424, 2001.
[21] K. V. R. Kanth, S. Ravada, and D. Abugov. Quadtree and r-tree indexes in oracle spatial: a comparison using GIS data. In *Proc. of SIGMOD*, pages 546–557, 2002.
[22] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. A. Boncz, T. Neumann, and A. Kemper. Approximate geospatial joins with precision guarantees. In *Proc. of ICDE*, pages 1360–1363, 2018.
[23] A. Klinger. Patterns and search statistics. In *Optimizing methods in statistics*, pages 303–337. Elsevier, 1971.
[24] H. Lang, A. Kipf, L. Passing, P. A. Boncz, T. Neumann, and A. Kemper. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. In *DaMoN*, pages 5:1–5:8, 2018.
[25] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proc. of ICDE*, pages 38–49, 2013.
[26] *Magellan: Geospatial Analytics Using Spark.* https://github.com/harsha2010/magellan.
[27] J. A. Orenstein. Redundancy in spatial databases. In *Proc. of SIGMOD*, pages 295–305, 1989.
[28] V. Pandey, A. Kipf, T. Neumann, and A. Kemper. How good are modern spatial analytics systems? *PVLDB*, 11(11):1661–1673, 2018.
[29] V. Pandey, A. Kipf, D. Vorona, T. Mühlbauer, T. Neumann, and A. Kemper. High-performance geospatial analytics in HyPerSpace. In *Proc. of SIGMOD*, pages 2145–2148. ACM, 2016.
[30] PostGIS: Spatial and geographic objects for PostgreSQL. http://postgis.net/.
[31] J. Roth. The extended split index to efficiently store and retrieve spatial data with standard databases. In *Proc. of IADIS*, pages 85–92, 2009.
[32] Google S2 library. http://s2geometry.io/.
[33] S2Geometry Basic Types. http://s2geometry.io/devguide/basic_types.html.
[34] Google S2ShapeIndex. https://s2geometry.io/devguide/s2shapeindex.
[35] D. Sidlauskas, S. Chester, E. Tzirita Zacharatou, and A. Ailamaki. Improving spatial data processing by clipping minimum bounding boxes. In *Proc. of ICDE*, pages 425–436, 2018.
[36] STX B+-tree. http://panthema.net/2007/stx-btree/.
[37] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref. LocationSpark: A distributed in-memory data management system for big spatial data. *PVLDB*, 9(13):1565–1568.
[38] TLC Trip Record Data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.
[39] E. Tzirita Zacharatou, H. Doraiswamy, A. Ailamaki, C. T. Silva, and J. Freire. GPU rasterization for real-time spatial aggregation over arbitrary polygons. *PVLDB*, 11(3):352–365, 2017.
[40] Uber geofence. https://eng.uber.com/go-geofence/.
[41] F. van Diggelen and P. Enge. The world's first gps mooc and worldwide laboratory using smartphones. In *Proc. of ION GNSS+*, pages 361–369, 2015.
[42] D. Vorona, A. Kipf, T. Neumann, and A. Kemper. DeepSPACE: Approximate geospatial query processing with deep learning. In *Proc. of SIGSPATIAL*, pages 500–503, 2019.
[43] C. Winter, A. Kipf, T. Neumann, and A. Kemper. GeoBlocks: A query-driven storage layout for geospatial data. *CoRR*, abs/1908.07753, 2019.
[44] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *Proc. of SIGMOD*, pages 1071–1085, 2016.
[45] S. You, J. Zhang, and L. Gruenwald. Parallel spatial query processing on gpus using r-trees. In *SIGSPATIAL Workshops*, pages 23–31, 2013.
[46] J. Yu, J. Wu, and M. Sarwat. Geospark: a cluster computing framework for processing large-scale spatial data. In *Proc. of SIGSPATIAL*, pages 70:1–70:4, 2015.
[47] T. Zäschke, C. Zimmerli, and M. C. Norrie. The ph-tree: a space-efficient storage structure and multi-dimensional index. In *Proc. of SIGMOD*, pages 397–408, 2014.
[48] J. Zhang and S. You. Speeding up large-scale point-in-polygon test based spatial join on gpus. In *Proc. of SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 23–32. ACM, 2012.
[49] G. Zimbrao and J. M. de Souza. A raster approximation for processing of spatial joins. In *Proc. of VLDB*, pages 558–569, 1998.