Approximate Geospatial Joins with Precision Guarantees

Andreas Kipf*, Harald Lang*, Varun Pandey*, Raul Persa*, Peter Boncz†, Thomas Neumann*, Alfons Kemper*

*Technical University of Munich
Munich, Germany
{first.last}@in.tum.de

†Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
boncz@cwi.nl

***Abstract*—Geospatial joins are a core building block of connected mobility applications. An especially challenging problem are joins between streaming points and static polygons. Since points are not known beforehand, they cannot be indexed. Nevertheless, points need to be mapped to polygons with low latencies to enable real-time feedback.**

We present an approximate geospatial join that guarantees a user-defined precision. Our technique uses a quadtree-based hierarchical grid to approximate polygons and stores these approximations in a specialized radix tree. Our approach can perform up to several orders of magnitude faster than existing techniques while providing sufficiently precise results for many applications.

I. Introduction

Connected mobility companies need to process vast amounts of location data in near real-time to run their businesses. For instance, Uber needs to join locations of passenger requests with a set of predefined polygons to display available products (e.g., Uber X) and to enable dynamic pricing[1]. Another example are traffic use cases where the positions of vehicles need to be joined with street segments to enable real-time traffic control. With a future of connected (and possibly self-driving) cars and drones, high-performance geospatial joins will become a key feature for increasingly demanding applications.

Geospatial joins have been studied for decades [1] and many of these algorithms follow the traditional *filter and refine* approach. A common technique is to index minimum bounding rectangles (MBRs) of polygons in an R-tree. For each point, the R-tree is probed and candidate polygons are refined using expensive point-in-polygon (PIP) tests.

True hit filtering partially avoids expensive refinements by identifying actual join pairs already in the filter phase [2]. This is achieved by approximating the interior of individual polygons using inner circles or rectangles. When a point falls into an interior approximation of a polygon, it is known to be within the polygon.

Building on top of these key ideas we present an improved algorithm that combines true hit filtering with quadtree-based hierarchical grids [3], [4]. This is in contrast to existing implementations of true hit filtering that use inner rectangles [5] or non-hierarchical grids (e.g., Spark Magellan[2]). In our approach, arbitrary geometrical shapes are translated into *sets of hierarchical grid cells* which are used to approximate geometries and their interiors. To support efficient queries, we store these approximations in a specialized in-memory radix tree (trie) named *Adaptive Cell Trie* (ACT).

Thereby it improves upon the state-of-the-art as follows:

- ACT improves the ratio of true hits by covering the majority of the interior area of polygons using interior cells.
- It achieves a lower false positive rate for candidate hits by using tight approximations of the boundary areas of polygons.
- Further, ACT can entirely avoid the expensive refinement phase by refining cells in the boundary areas until a user-defined precision of up to a few centimeters is guaranteed (more than enough for GPS data).
- Finally, it outperforms existing implementations by up to two orders of magnitude, since index lookups only rely on (a few) basic integer arithmetics and bitwise operations.

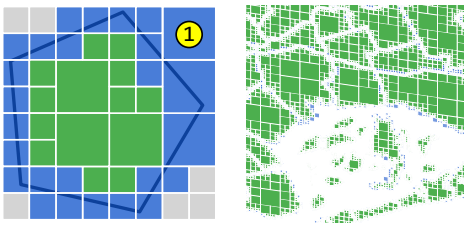
Of course, these improvements come at the cost of higher memory consumption. However, we argue that with the large main-memory capacities of modern hardware we can afford to maintain fine-grained index structures purely in main memory. Depending on the employed grid, the approximations can be very accurate. Our reference implementation supports very high resolutions, up to a few centimeters.

To the best of our knowledge, this work is the first to completely avoid the refinement phase while providing precision guarantees. Given the fact that the processing of lat/lng coordinates is inherently imprecise due to their representation as floating point numbers and that GPS positions (typically obtained by smartphones) approximately have a 5 m accuracy under open sky [6], we argue that trading off precision with performance is a valid choice for many geospatial applications.

Our approach is also applicable to situations with strict memory constraints. If ACT cannot guarantee the desired precision given a certain memory budget, the refinement

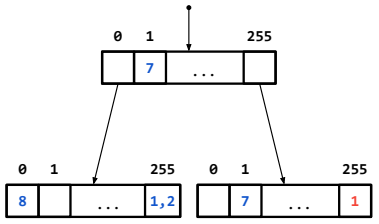
[1]https://eng.uber.com/go-geofence/

[2]https://github.com/harsha2010/magellan



(a) Coverings (b) Super covering

Fig. 1: A covering and an interior covering of an individual polygon (a) and a super covering of neighborhoods in NYC's Jamaica Bay (b) — blue = covering, green = interior



(a) Adaptive Cell Trie

offset	true	candidate
0	{5}	{3, 1}
1	{7, 2}	{8}
...	...	...

(b) Lookup table

Fig. 2: Adaptive Cell Trie and the lookup table. Key parts (bit sequences) are marked in black and values are marked in blue (single or double payloads) and red (offsets).

phase clearly cannot be omitted. Our solution is to adaptively alter the trie structure based on the distribution of query points to provide higher precision where it is actually needed. Thus, the probability for true hits increases, false positives are reduced, and consequently, expensive refinements can be avoided. Due to space constraints, in the remainder of this paper we present the basic functionality of ACT assuming no memory constraints. We will cover adaptive versions of ACT in future work.

II. Approach

The high level idea is to compute fine-grained approximations of sets of polygons and store them in ACT, a specialized in-memory radix tree. Our index is parameterized with an application-dependent precision bound[3] that must be guaranteed.

We begin by computing approximations of individual polygons. Figure 1a shows a covering and an interior covering of an individual polygon marked in blue and green, respectively. 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 $\sqrt{2} * a$ (with a being the side length of the cell) to the polygon.

To avoid expensive PIP tests, we can treat all points contained in covering cells as (approximate) hits. Thereby we introduce false positives. However, as described above, the distance of false positives from the polygon is bounded by the diagonal of the largest covering cell. Thus, to satisfy a desired precision, we can refine the largest covering cells until they are sufficiently small.

In our implementation, we use Google S2 to compute the individual coverings. Note that our approach does not depend on S2 and in fact works with any other quadtree-based hierarchical grid where each (implicit) quadtree node [7] 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 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.

A key feature of S2 is that it can represent each cm^2 on Earth using a 64 bit unsigned integer (cell id). We refer the reader to the S2 website[4] for more details about the library.

Once the coverings of every polygon have been computed, we merge these individual coverings into a *super covering* that represents all polygons. This step involves removing duplicate cells and resolving conflicts between overlapping cells. The latter may require additional refinement steps and potentially increases the total number of cells.

Figure 1b shows a super covering of neighborhoods in NYC's Jamaica Bay. Covering and interior cells are again marked in blue and green, respectively. Most of the shown area is either covered by interior cells or by no cells at all. Only in the unlikely event that a query point hits polygon boundaries, we may experience false positives.

Each cell in the super covering references a set of polygons. A reference consists of a *polygon identifier* and an *interior flag* that indicates whether the cell is a covering or an interior cell of the respective polygon. To allow for efficient lookups, we store the cells in ACT and their polygon references in a separate lookup table. Both data structures are designed for in-memory processing and are optimized to only cause a minimal number of cache misses. For performance and space efficiency reasons, not every indexed cell points to an entry of the lookup table. In most cases, cells reference one or two polygons. Therefore, we *inline* the polygon identifiers in the trie structure to eliminate additional indirections.

Adaptive Cell Trie: During join processing, we only perform *prefix* lookups, i.e., we search for the cell ids that share a common prefix with the cell id of the query point[5] to check whether the query point is contained in one of the indexed cells. A radix tree thus is the ideal data structure for our needs. For example, it is more space-efficient than a (sorted) vector, since it avoids redundantly storing common prefixes (in a trie, the path to a leaf node implicitly defines the key). Likewise, lookups are in $O(k)$ with k being the key length as opposed to the $O(log\, n)$ runtime complexity of binary search that could

[3]The maximum distance between the partners of a false positive join pair.

[4]http://s2geometry.io/

[5]The query point is translated into a cell on the most fine-grained grid level.

be used on a sorted vector. We refer the reader to the work of Zäschke at al. for another example of storing cells in a trie structure [8].

Since we are interested in being highly selective and therefore need to index many cells, the trie structure usually exceeds cache size (cf., Section III). Thus, traversing the tree potentially results in many cache misses.

Let k_{avg} be the average key length and f be the fanout of the tree. Then the average costs c_{avg} of a lookup can be estimated as follows:

$$c_{avg} = \lceil k_{avg}/log_2(f) \rceil * \textit{costs per node access}$$

The number of node accesses is bounded by the maximum key length k_{max}, which is 60 when using 30 quadtree levels (like in our case/implementation).

ACT uses a default fanout of 256 (= 8 bits). Thus, every level in the tree represents four grid levels (every level is encoded with two bits). This has the side effect that we can only index cells at certain levels.

Let g be the cell level granularity of the tree. Then the following holds for indexed cells:

$$level_{cell} \bmod g = 0$$

Thus, we need to denormalize[6] cells upon insertion and replicate their payloads.

While a fanout of 256 results in sparsely occupied trie nodes and thus in a high space consumption, 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 $\lceil 60/log_2(256) \rceil = 8$. In practice, a lower k_{max} is often sufficient. For example, $k_{max} = 48$ allows for indexing cells up to level 24 which limits the error of false positives to less than 1 m (in our implementation) and reduces the number of maximum node accesses to 6.

Figure 2a illustrates the structure of ACT. Values (payloads or offsets) can be found in any node at any level of the tree. Every node consists of a fixed-sized array of 256 entries of 8 byte pointers. These pointers are *tagged*. By default, all entries point to a sentinel node indicating a false hit. Such a tagged entry can be:

- An 8 byte pointer to a child or the sentinel node
- An inlined payload (a 31 bit value)
- Two inlined payloads (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 payload, we differentiate between a true hit and a candidate hit using the least significant bit of the 31 bit payload. Thus, we can effectively only store 30 bit payloads (i.e., index up to 2^{30} polygons).

Lookup table: When a cell references more than two polygons, the tree contains an offset into a lookup table. Cells often reference the same set of polygons. To avoid redundancy, we therefore only store *unique* polygon reference sets. Figure 2b shows an example of a lookup table. The reference sets are split into two parts, a set with true hits and a set with candidate hits. Both sets contain polygon identifiers. When a cell references at most two polygons, we inline its payloads into the tree (as described above). 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.

A lookup in ACT returns at most one cell that maps to a set of polygon references. In contrast to a search in a binary tree, a search in a radix tree is comparison-free. This means that we do not compare the value of the search key to the value(s) stored in the current node. We only need to extract the relevant bits of the search key and jump to the corresponding offset. However, we do comparisons to differentiate between pointers and inlined payloads.

III. Evaluation

We evaluate our approach on a server-class machine that is equipped with two 14-core Intel Xeon E5-2680 v4 CPUs and 256 GB DDR4 RAM. All experiments are conducted on a single socket to eliminate NUMA effects. We join 1 B points from the NYC taxi dataset[7] and against NYC's boroughs (5 polygons), neighborhoods (289 polygons), and census blocks (39,184 polygons) and count the number of points per polygon. While there are only five boroughs, their polygons are significantly more complex. We use GCC version 5.4.0 with `O3` and `march=core-avx2` flags in all experiments.

As a baseline of comparison, we index the MBRs of the polygons in the boost R-tree (1.60.0) and measure its lookup performance *without* refining candidates. For each returned candidate, we simply increase the counter of the respective polygon. We use the splitting strategy `rstar` with a maximum of 8 elements per node which performs best in all workloads. The R-tree consumes 376 bytes, 27.9 kB, and 3.49 MB for the boroughs, neighborhoods, and census blocks dataset, respectively. Note that this approach does not guarantee any precision and only serves as a baseline for lookup performance of in-memory spatial indexes.

The performance of our approach is dominated by the costs for the ACT node accesses and the aggregation (counting the number of points per polygon). To better understand the results, we therefore first analyze the space consumption of our index.

Table I shows different metrics of our index for the three polygon datasets with 60 m, 15 m, and 4 m precision. While the two 60 m indexes of the boroughs and the neighborhoods dataset fit into the cache, all other indexes significantly exceed cache size. For the census blocks dataset, ACT with 4 m precision consumes 1.21 GB of memory. Note that even when the number of indexed cells increases, the size of ACT does

[6]Denormalizing a cell to a given level means replacing the cell with all of its descendant cells at that level.

[7]http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

TABLE I: Metrics of our index

	boroughs			neighborhoods			census		
precision [m]	60	15	4	60	15	4	60	15	4
indexed cells [M]	0.09	1.33	21.1	0.16	0.98	14.1	8.51	8.97	39.8
ACT [MB]	1.42	170	170	25.7	140	140	1162	1206	1206
lookup table [MB]	0.00	0.00	0.00	0.01	0.01	0.01	1.33	1.33	1.41
build individual coverings [s]	1.43	18.3	303	0.37	1.22	14.7	6.28	8.90	47.7
build super covering [s]	0.31	1.81	17.9	0.62	1.82	14.2	40.7	40.2	88.2

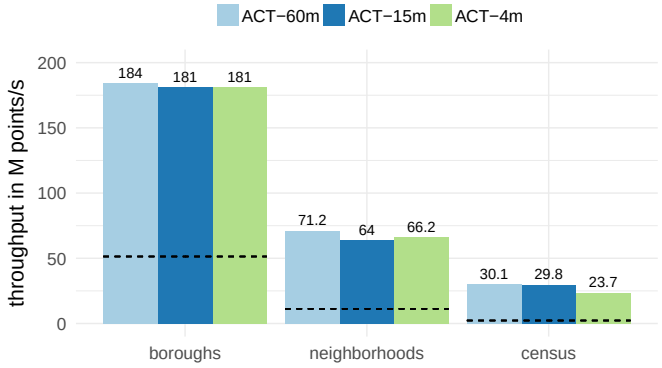


Fig. 3: Single-threaded throughput of our approach with varying precision for the three polygon datasets. The dashed lines indicate the lookup performance of the boost R-tree.

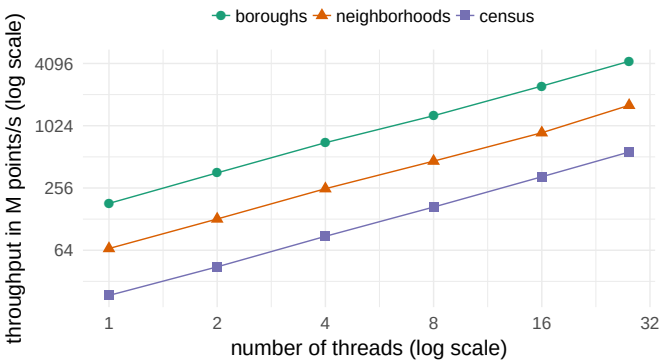


Fig. 4: Scalability of our approach with 4 m precision with a peak throughput of 4.30 B points/s for boroughs

not necessarily increase. This is an artefact of the high fanout of our radix tree. Since we are addressing the case of static polygons, we only minimally optimized the build phase in our implementation. While the computation of the individual coverings is parallelized over the number of polygons, the construction of the super covering runs serially.

Figure 3 shows the single-threaded throughput of our approach with varying precision compared to the baseline (dashed lines). `ACT-60m` achieves a throughput of 184 M points/s for the boroughs dataset. With a higher number of polygons in the neighborhoods and census datasets, the throughput of `ACT-60m` decreases. The more precise indexes `ACT-15m` and `ACT-4m` show similar performance numbers. For the boroughs dataset, `ACT-4m` achieves almost the same performance as `ACT-60m`. The reason for this is that boroughs have large interior areas and thus points are very likely to hit (coarse-grained) interior cells which are indexed in upper (cached) ACT nodes (due to their short cell ids). Compared to the baseline, `ACT-4m` achieves a 3.54x, 5.86x, and 10.3x higher performance for the boroughs, neighborhoods, and census blocks dataset, respectively. The fact that this factor increases for the larger datasets shows that our approach scales better with the number of polygons.

Finally, we evaluate the scalability of our technique. We use `ACT-4m` for this experiment since it significantly exceeds the cache size of our evaluation machine for all datasets. Figure 4 shows the results. Our approach scales well for all three datasets with the number of physical cores and even with the number of hyperthreads. The fact that an oversubscription of cores has a positive performance impact shows that our technique is bound by memory access latencies and having more threads than physical cores can hide these latencies.

IV. Conclusions

We have presented an approximate geospatial join that guarantees a user-defined precision. Our technique uses a quadtree-based hierarchical grid to approximate polygons represented by a specialized radix tree. 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.

Acknowledgments

This work has been sponsored by the German Federal Ministry of Education and Research (BMBF) grant FASTDATA 01IS12057. This work is further part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs and Media, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

References

[1] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM Trans. Database Syst.*, vol. 32, no. 1, p. 7, 2007.
[2] T. Brinkhoff, H. Kriegel, R. Schneider, and B. Seeger, "Multi-step processing of spatial joins," in *Proc. of SIGMOD*, 1994, pp. 197–208.
[3] A. Klinger, "Patterns and search statistics," in *Optimizing methods in statistics*. Elsevier, 1971, pp. 303–337.
[4] G. M. Hunter and K. Steiglitz, "Operations on images using quad trees," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 1, no. 2, pp. 145–153, 1979.
[5] K. V. R. Kanth and S. Ravada, "Efficient processing of large spatial queries using interior approximations," in *Proc. of SSTD*, 2001, pp. 404–424.
[6] F. van Diggelen and P. Enge, "The worlds first gps mooc and worldwide laboratory using smartphones," in *Proc. of ION GNSS+ 2015*, 2015, pp. 361–369.
[7] I. Gargantini, "An effective way to represent quadtrees," *Commun. ACM*, vol. 25, no. 12, pp. 905–910, 1982.
[8] T. Zäschke, C. Zimmerli, and M. C. Norrie, "The ph-tree: a space-efficient storage structure and multi-dimensional index," in *Proc. of SIGMOD*, 2014, pp. 397–408.