Progressive, Holistic Geospatial Interlinking

George Papadakis[1], Georgios Mandilaras[1], Nikos Mamoulis[2], Manolis Koubarakis[1]
[1]National and Kapodistrian University of Athens, Greece {gpapadis,gmandi,koubarak}@di.uoa.gr
[2]University of Ioannina, Greece nikos@cs.uoi.gr

ABSTRACT

Geospatial data constitute a considerable part of Semantic Web data, but at the moment, its sources are not sufficiently interlinked with topological relations in the Linked Open Data cloud. Geospatial Interlinking aims to cover this gap through space tiling techniques, which significantly restrict the search space. Yet, the state-of-the-art techniques operate exclusively in a batch manner that produces results only after processing the entire input datasets. In each run, they are also restricted to searching for an individual topological relation, even though most operations are common for the 10 main relations. In this work, we address both issues: we introduce a batch algorithm that simultaneously computes all topological relations and define the task of Progressive Geospatial Interlinking, which produces results in a pay-as-you-go manner when the available computational or temporal resources are limited. We propose two progressive algorithms and explain how they can be adapted to massive parallelization with Apache Spark. We conduct a thorough experimental study over a six large, real datasets, demonstrating the superiority of our techniques over the current state-of-the-art.

ACM Reference Format:
George Papadakis[1], Georgios Mandilaras[1], Nikos Mamoulis[2], Manolis Koubarakis[1]. 2021. Progressive, Holistic Geospatial Interlinking. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3449850

1 INTRODUCTION

The Web abounds in huge datasets of geospatial data, such as OpenStreetMap[1] and the US Census Bureau TIGER files[2]. Alone, Geographica 2.0 [8] has gathered almost half a billion RDF triples from various open data sources, such as the CORINE Land Cover dataset[3], while the spatial knowledge base LinkedGeoData conveys more than 3 billion geographic entities (*geometries* in the following) and 20 billion RDF triples[4].

To leverage these voluminous sources of geospatial data, we need to capture all important topological relations between their geometries, according to the DE-9IM model [2, 3, 6]. These relations are indispensable for crucial applications like the Icewatch project[5], which gathers in-situ observational data about icebergs from ships navigating the arctic and associates them with geospatial information extracted from satellite images offered by Earth observation programmes, such as the US Landsat program[6] and the EU Copernicus Programme[7]. Due to the effects of climate change, this information needs to be frequently updated in order to ensure safe ship navigation.

Moreover, applications involving reasoning [11, 13], question answering [27] or simply running GeoSPARQL queries over geospatial data [17] call for increasing the interlinking between the datasets of the *Linked Open Data* (LOD) cloud. At the moment, though, the geospatial data is underrepresented in the LOD Cloud[8]: even though it corresponds to almost 20% of the LOD cloud triples, only 7% of the triples linking different datasets pertain to geometries [14].

Such applications can be facilitated by *Geospatial Interlinking* [19, 20, 23], i.e., the task of automatically finding topological relations between all input geometries. However, Geospatial Interlinking may have to compare every geometry with all others, thus having a quadratic time complexity with respect to the input geometries. The computational cost of verifying a single topological relation is also high: each geometry is converted into a labelled topology graph, with a vertex for each point and an edge for each pair of consecutive points[9]. The two graphs are then merged in order to check a topological relation between the respective geometries in a way that considers their interior, boundary and exterior. The time complexity is approximately $O(N \cdot \log N)$, where N is the number of edges in the merged graphs [1]. Therefore, existing Geospatial Interlinking approaches [19, 20, 23] scale to large volumes of data by avoiding the brute-force approach of verifying all geometry pairs. The number of required computations is reduced without sacrificing effectiveness (i.e., the identified links between the input geometries) by operating in two steps:

1) *Filtering* drastically reduces the number of candidate geometry pairs, through *space tiling*, which imposes a uniform grid over the data space and assigns each geometry to all tiles that intersect its Minimum Bounding Rectangle (MBR). This is illustrated in Figure 1.

2) *Verification* is applied to all pairs of geometries, which co-occur in at least one tile, to confirm their topological relation.

Current state-of-the-art. *Silk-spatial* [23] employs a *static* space tiling approach that defines a fixed EquiGrid on Earth's surface, independently of the input data. As a result, its tiles are usually coarse-grained, which means that the number of geometry pairs that are verified is too large, incurring a computational cost that is close to that of a brute-force approach [20]. This is partially ameliorated through massive parallelization on top of Apache Hadoop[10].

[1]https://www.openstreetmap.org
[2]http://spatialhadoop.cs.umn.edu/datasets.html
[3]https://land.copernicus.eu/pan-european
[4]http://linkedgeodata.org/About
[5]https://icewatch.met.no
[6]https://www.usgs.gov/core-science-systems/nli/landsat
[7]https://www.copernicus.eu/en
[8]https://lod-cloud.net
[9]https://locationtech.github.io/jts/jts-faq.html
[10]https://hadoop.apache.org

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.
WWW '21, April 19–23, 2021, Ljubljana, Slovenia
© 2021 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC-BY 4.0 License.
ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3449850

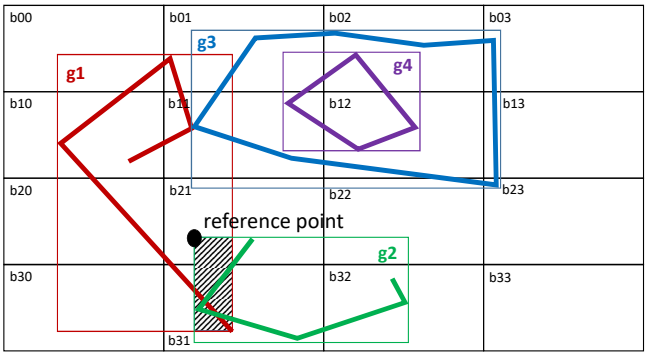

Figure 1: The space tiling approach for four geometries, where g_1 is a LineString that `intersects` LineString g_2 and `touches` Polygon g_3, which `contains` Polygon g_4. The shaded area corresponds to the intersection of the MBRs of g_1 and g_2. Its top left corner is used as a reference point to avoid verifying the same pair more than once.

RADON [20] improves on Silk-spatial by building *dynamic, fine-grained* tiles: in each dimension, the extent of the tiles has a length equal to the average extent of the geometry MBRs in that dimension. Given that every geometry is assigned to all tiles intersecting its MBR, the contents of the resulting tiles are overlapping and, thus, abound in *redundant* geometry pairs, i.e., pairs of intersecting geometries that co-occur in multiple tiles. RADON maintains in main memory a hash-table with all geometry pairs verified so far, which is used to avoid verifying redundant pairs more than once. Its tile-centric approach also maintains all input data in main memory, thus yielding very high space requirements.

Regarding Verification, both approaches limit it to a single topological relation, even though the same grid is used for all relations. This means that if another relation needs to be examined over the same data, the entire algorithm is repeated from scratch.

Note also that both approaches operate in a batch (i.e., budget-agnostic) way that consumes the input datasets in no particular order. Thus, they cannot schedule their processing to fully exploit high-end platforms with extreme capabilities for massive parallel processing, but limited availability, such as Amazon Web Services[11].

Contributions. In this work, we go beyond the current state-of-the-art in a number of ways:

1) In Section 3, we define *Holistic Geospatial Interlinking* as the task of simultaneously computing all non-trivial topological relations. We also define *Progressive Holistic Geospatial Interlinking* as the task of computing all non-trivial topological relations in a pay-as-you-go manner that prioritizes the verification of geometry pairs based on heuristics.

2) In Section 4, we introduce *GIA.nt*, a batch algorithm for Holistic Geospatial Interlinking that reduces RADON's space requirements by at least 50% and its run-time almost by an order of magnitude.

3) In Section 5, we introduce several weighting schemes that can be used to define the optimal order of geometry pairs for Progressive Holistic Geospatial Interlinking.

4) In Section 6, we explain how RADON and GIA.nt can be adapted to address Progressive Holistic Geospatial Interlinking. Compared to batch RADON, the current state of the art, Progressive GIA.nt is able to reduce the time required to detect the vast majority of topological relations by a whole order of magnitude.

5) In Section 7, we parallelize (Progressive) GIA.nt on top of Apache Spark (http://spark.apache.org) so as to scale sublinearly to voluminous data, interlinking 190M geometries within minutes.

7) In Section 8, we evaluate all approaches through a thorough experimental analysis that involves six real, large-scale datasets.

[11] https://aws.amazon.com

2 RELATED WORK

As already discussed, Silk-spatial and RADON are the main algorithms for Geospatial Interlinking. The generic schema matching system *AgreementMaker* [4] has also been adapted to this problem, outperforming Silk-spatial in terms of time efficiency [19]. However, the thorough experimental evaluation in [19] demonstrates that RADON is significantly faster than both Silk-spatial and AgreementMaker, constituting the state-of-the-art approach. This verifies the experimental results of the "Spatial" track of the OM-2019 workshop (http://om2019.ontologymatching.org).

Note that Geospatial Interlinking differs from traditional spatial joins, which mainly look for geometry pairs that intersect or pairs of nearby points. Instead, Geospatial Interlinking examines a variety of topological relations, as explained in Section 3. Partitioning-based spatial join algorithms [10] like PBSM [16] are mostly appropriate when the geometries are relatively small compared to the tiles (as in the coarse-grained grid of Silk-spatial). On the other hand, when using a fine-grained grid (as in RADON), each geometry typically participates in many, small tiles (e.g., in the example of Figure 1).

Another important task is the preprocessing of input geometries in order to correct digitization errors and sliver polygons, either manually or by leveraging ontologies [18]. This data cleaning is orthogonal to Geospatial Interlinking, as it can be applied after preprocessing, directly to the refined data.

More importantly, we focus on pay-as-you-algorithms that try to optimize the processing order of geometry pairs within a limited budget (in terms of computational resources). Previous work on progressive computation of spatial joins focuses on stream processing and is not relevant to our problem. Coarse object approximations are used in [12] to avoid accessing and buffering detailed representations as much as possible. A progressive version of PBSM for streaming data is proposed in [24], using statistics to determine which contents to keep in the memory buffer and which in disk during evaluation, in order to maximize the join throughput.

Finally, a problem related to this work is *Progressive Entity Resolution* [22, 26], where the goal is to detect matches, i.e., entity profiles describing the same real-world object, in a pay-as-you-go manner. For example, *Progressive Sorted Neighborhood* [26] orders the input entities in alphabetical order of their associated blocking keys. Then, a window of size $w = 1$ slides over the sorted list of entities to compare those in consecutive positions. After processing the entire list, the window size is incremented ($w = 2$) and the processing starts from the top of the list and so on and so forth. A schema-agnostic version of this approach is presented in [22]: every entity is associated with multiple blocking keys and, thus, with multiple positions in the sorted list. Weighting schemes are defined to order the distinct pairs of entities according to their co-occurrence frequency in the sliding window of the current size.

3 PRELIMINARIES

In this work, we are interested in geometries that consist of interior, boundary and exterior (i.e., all points that are not part of the interior or the boundary). They are distinguished into two main types [19]: (i) *LineStrings*, which constitute one-dimensional geometries formed by a sequence of points and the line segments that connect consecutive points (e.g., g_1 and g_2 in Figure 1), and (ii) *Polygons*, which in the simple case are two-dimensional geometries formed by a sequence of points where the first one coincides with the last one (e.g., g_3 and g_4 in Figure 1).

For two geometries of these types, s and t, the *Dimensionally Extended nine-Intersection Model* (DE-9IM) [2, 3, 6] defines 10 main topological relations:

1) `Intersects`(s,t) suggests that s and t share at least one point in their interior or boundary.

2) `Contains`(s,t) means that s lies inside t such that only their interiors intersect.

3) `Within`(s,t) means that t `Contains` s.

4) `Covers`(s,t) indicates that s lies inside t such that their interiors or their boundaries intersect.

5) `Covered_by`(s,t) means that t `Covers` s.

6) `Equals`(s,t) means that the interiors of s and t intersect, but no point of s intersects the exterior of t and vice versa.

7) `Touches`(s,t) indicates that the two geometries share at least one point, but their interiors do not intersect.

8) `Crosses`(s,t) indicates that the two geometries share some but not all interior points and that the dimension of their intersection is smaller than that of at least one of them (see Section 3.1 for the definition of dimension).

9) `Overlap`(s,t) differs from `Crosses`(s,t) in that the two geometries have the same dimension, and so does their intersection.

10) `Disjoint`(s,t) designates that s and t share no interior or boundary point.

See Figure 1 for examples of these topological relations. Note that in the following, we disregard the `Disjoint` relation; unlike all other topological relations, which scale linearly with the size of the input data, `Disjoint` scales quadratically, because the vast majority of pairs typically pertains to unrelated geometries (see Table 2). When the input comprises few million geometries, it is impossible to generate trillions of `Disjoint` links. Besides, `Disjoint` is not interesting, because it provides no practical linking between entities. It can be inferred, though, from `Intersects` [20]: geometries that are not associated with the latter relation are disjoint.

3.1 Problem Definition

Given that each topological relation r is a predicate evaluating to true or false, Geospatial Interlinking is defined as [19, 20, 23]:

Definition 1 (Geospatial Interlinking). *Given a source dataset S, a target dataset T and a topological relation r, discover the set of links $L_r = \{(s,r,t)|s \in S \wedge t \in T \wedge r(s,t)\}$.*

In the context of Linked Data, the goal is to estimate all topological relations (excluding `Disjoint`) between the source and the target datasets. Yet, as already discussed, previous Geospatial Interlinking approaches look for a single topological relation in each run [19, 20, 23]. This is counter-intuitive, since the same Filtering step is applied for all topological relations in R. Most importantly, all

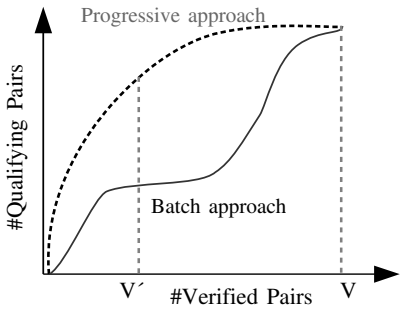

Figure 2: Evolution of the number of topologically related pairs (vertical axis) as more pairs are verified (horizontal axis) using a progressive and a batch approach.

topological relations can be derived with simple logical conditions from the *Intersection Matrix* [19], which is defined as:

$$IM(s,t) = \begin{bmatrix} dim(I(s) \cap I(t)) & dim(I(s) \cap B(t)) & dim(I(s) \cap E(t)) \\ dim(B(s) \cap I(t)) & dim(B(s) \cap B(t)) & dim(B(s) \cap E(t)) \\ dim(E(s) \cap I(t)) & dim(E(s) \cap B(t)) & dim(E(s) \cap E(t)) \end{bmatrix},$$

where dim denotes dimension of the intersection $\cap$ of the interior I, boundary B, and exterior E of the geometries s and t. In case of an empty intersection, dim is -1 or F (False), while for a non-empty intersection, dim is equal to 0 if the intersection is a point, 1 if it is a line segment and 2 if it is an area. The values {0,1,2} are collectively represented by T (True).

In this context, every topological relation can be defined as a logical condition on the values of the intersection matrix. For example, `Within` is defined as $IM(0,0) = T \wedge IM(0,2) = F \wedge IM(1,2) = F$ or equivalently as $\begin{bmatrix} T & * & F \\ * & * & F \\ * & * & * \end{bmatrix}$. Note, though, that we should avoid computing redundant relations. For example, if `Contains`(s,t) holds, then `Within`(t,s) is always true [19]. Thus, it suffices to add only the former as an explicit statement to the LOD cloud. The same applies to `Covers`(s,t) and `CoveredBy`(t,s).

Overall, Geospatial Interlinking can be restricted to identifying the above topological relations 1 to 9, which we call *non-trivial*. On this basis, we can minimize its cost by redefining it as follows:

Definition 2 (Holistic Geospatial Interlinking). *Given a source dataset S, a target one T, and the set of non-trivial topological relations R, derive the set of links $L_R = \{(s,r,t)|s \in S \wedge t \in T \wedge r \in R \wedge r(s,t)\}$ from the Intersection Matrix of geometry pairs.*

As explained above, this problem is addressed in two steps: first, Filtering applies space tiling to reduce the computational cost to geometries with MBRs co-occurring in one or more tiles. Then, Verification computes the Intersection Matrix of the candidate pairs produced by Filtering. Among them, Verification is the bottleneck, due to its high computational cost, i.e., $O(N \cdot \log N)$, where N is the total number of edges in the corresponding topological graphs [1].

Progressive Geospatial Interlinking. In this work, we also examine methods that solve the Geospatial Interlinking task in a progressive, i.e., pay-as-you-go, manner, when we have limited time or computational resources. Without loss of generality, we assume that the available resources for Progressive Geospatial Interlinking are defined in terms of the number of geometry pairs that are

actually verified. We call this number *budget* (*BU*). With minor modifications, our definitions and algorithms can be adapted to a temporal limit that specifies the available running time.

Assuming that a batch approach verifies V pairs, progressive methods must satisfy two requirements [26], as shown in Figure 2:

1) *Same Eventual Quality.* The results produced after V verifications, by a progressive and a batch approach should be identical, i.e., the progressive approach should eventually produce the same set of links as the batch approach.

2) *Improved Early Quality.* If a progressive and a batch approach were applied to the same datasets, S and T, and terminate after $V' = BU \ll V$ verifications, the former should detect significantly more *qualifying* geometry pairs, which satisfy at least one of the topological relations in R.

The second requirement suggests that we can assess the relative performance of progressive methods by the rate of producing results as more pairs are verified. We actually define *Progressive Geometry Recall* (PGR) as the rate of detecting qualifying geometry pairs and quantify it by the area under the curve that is formed by the corresponding lines in Figure 2. The larger this area is, the earlier the interlinked pairs are detected or more relations are computed, and the more effective is the progressive method. We formalize this measure as $PGR(R) = \sum_{i=1}^{|P|} p_Q^i / |P_Q^{BU}|$, where $P \subseteq S \times T$ is the set of candidate geometry pairs, which pass the Filtering step, $|P|$ is its size (i.e., the total number of candidate pairs), $P_Q^{BU} \subseteq P$ is the set of qualifying geometry pairs within the given budget BU, and p_Q^i is the total number of qualifying geometry pairs that have already been detected when processing the i^{th} candidate pair. PGR takes values in $[0, 1]$, with higher values indicating higher effectiveness.

In this context, we formalize pay-as-you-go interlinking as:

Definition 3 (Progressive, Holistic Geospatial Interlinking). *Given a source dataset S, a target one T, the set of non-trivial topological relations R and a limited budget of computational resources, maximize $PGR(R)$, given the available resources.*

4 BATCH ALGORITHM

We now introduce *Geospatial Interlinking At large* (**GIA.nt**), a novel batch algorithm that significantly improves RADON in terms of time and space requirements. Instead of indexing both source and target datasets, GIA.nt indexes only the source dataset. Moreover, instead of a tile-centric Verification, GIA.nt introduces the geometry-centric Verification: for every target geometry, GIA.nt aggregates the distinct source geometries in the tiles intersecting its MBR and processes them one by one, by computing their intersection matrix.

GIA.nt's functionality appears in Algorithm 1. Lines 1-12 apply Filtering to index the source dataset, which is set as the smallest one so as to minimize the memory footprint. In Line 2, the longitude and latitude granularity of tiles are defined as $\Delta_x = mean_{s \in S} MBR(s).width$ and $\Delta_y = mean_{s \in S} MBR(s).length$, resp., by adapting RADON's approach so that it considers only the source dataset (see Section 6.1 for more details). For each source geometry s (Line 3), GIA.nt estimates the diagonal corners of its MBR (Line 4) - the lower left point $(x_1(s), y_1(s))$ and the upper right point $(x_2(s), y_2(s))$. Using them along with Δ_x and Δ_y, it determines the tiles that intersect $MBR(s)$ and should contain s (Lines 5-11).

Algorithm 1: GIA.nt

```
   input : the source dataset S, a reader for the target one rd(T) &
           the set of non-trivial topological relations R
   output: the links L = {(s, r, t) | s ∈ S ∧ t ∈ T ∧ r ∈ R ∧ r(s, t)}
   /* Filtering step                                              */
1  I ← {} ;                          // Equigrid index structure
2  (Δx, Δy) ← defineIndexGranularity(S);
3  foreach geometry s ∈ S do
4      (x1(s), y1(s), x2(s), y2(s)) ← getDiagCorners(s);
5      for i ← ⌊x1(s) · Δx⌋ to ⌈x2(s) · Δx⌉ do
6          for j ← ⌊y1(s) · Δy⌋ to ⌈y2(s) · Δy⌉ do
7              I.addToIndex(i, j, s);
8              j ← j + 1;
9          end
10         i ← i + 1;
11     end
12 end
   /* Verification step                                           */
13 L ← {} ;                          // The set of detected links
14 while rd(T).hasNext() do
15     t ← rd(T).next() ; // The current target geometry
16     CS ← {} ;  // The set of candidate source geom.
17     (x1(t), y1(t), x2(t), y2(t)) ← getDiagCorners(t);
18     for i ← ⌊x1(t) · Δx⌋ to ⌈x2(t) · Δx⌉ do
19         for j ← ⌊y1(t) · Δy⌋ to ⌈y2(t) · Δy⌉ do
20             CS ← CS ∪ I.getTileContents(i, j);
21             j ← j + 1;
22         end
23         i ← i + 1;
24     end
25     foreach geometry s ∈ CS do
26         if intersectingMBRs(s, t) then
27             IM ← verify(s, t);
28             L ← L ∪ IM.getRelations();
29         end
30     end
31 end
32 return L;
```

To clarify how this space tiling approach works, assume that the longitude and latitude granularity are $\Delta_x = 3$ and $\Delta_y = 2$, respectively. For *POLYGON* ((19 60, 19 61, 14 61, 14 60, 19 60)), the diagonal corners of its MBR are defined by the lower left point (14, 60) and the upper right point (19, 61). As a result, this geometry will be placed in the tiles defined by $\lfloor 14/\Delta_x \rfloor = 4 \leq i \leq 5 = \lceil 14/\Delta_x \rceil$ and $\lfloor 60/\Delta_y \rfloor = 30 \leq j \leq 31 = \lceil 60/\Delta_y \rceil$.

GIA.nt's Verification is applied in Lines 13-31. The next target geometry $t \in T$ is read from the disk (Lines 14-15) and the tiles that would contain it are estimated, based on its MBR (Lines 17-24). For each tile, GIA.nt retrieves the source geometries it contains and places them in the local *set* of candidates C_S (Line 20). As a result, every source geometry that is likely to be related to t appears in C_S just once. Next, GIA.nt iterates over the geometries of C_S (Line

25) and those with an intersecting MBR (Line 26) are verified, by computing the corresponding intersection matrix IM (Line 27). The topological relations that are extracted from IM are added to the list of detected links L (Line 28), which is returned as output (Line 32).

GIA.nt's space complexity amounts to $O(|S| + |L|)$, which is at least 50% lower than RADON's $O(|S| + |T| + |L| + |V|)$, where V stands for the set of verified pairs. The reason is that GIA.nt does not maintain in memory both input datasets, but only the source one, which is the smallest by definition (i.e., $|S| \leq |T|$). The target geometries are read one by one from the disk; hence, the tiles, which are kept in memory, contain only source geometries – no target ones. GIA.nt also avoids all redundant pairs inherently, by maintaining a set with the distinct source geometries in the tiles of every target geometry. In contrast, RADON avoids redundant verifications by storing all verified pairs in memory, increasing its space complexity by $O(|V|)$.

GIA.nt is also much faster than RADON. For Filtering, its time complexity is $O(|S|)$, as it iterates once over the source geometries. Thus, it is at least 50% faster than RADON's $O(|S|+|T|)$, which goes through all input geometries to index them and to compute heuristics for switching inputs. For Verification, GIA.nt's time complexity is dominated by the number of candidate pairs, i.e., geometries with intersecting MBRs: $O(|(s,t) : MBR(S) \cap MBR(T)|)$. RADON's Verification cost amounts to $O(|R| \cdot |(s,t) : MBR(S) \cap MBR(T)|)$, since it repeats the same process for every topological relation, instead of simultaneously computing all of them, as GIA.nt does. Note also that unlike RADON, GIA.nt's Verification can be easily adapted to massive parallelization according to the MapReduce paradigm, since every target geometry can be processed locally in a cluster node, without the need to store in main memory all target geometries and all verified pairs.

5 WEIGHTING SCHEMES

The gist of progressive methods is to schedule the verification of geometry pairs in a way that maximizes the Progressive Geometry Recall (see Definition 3). To this end, they assign a weight to every pair that is analogous to its probability to satisfy a non-trivial relation: the higher the weight, the earlier the corresponding pair should be processed. Assuming that all geometry pairs bear the same computational cost, we consider *hit probability weighting schemes*, which produce a numerical estimate of how likely two geometries s and t are to satisfy a non-trivial topological relation, judging exclusively from the tiles that contain them. The more tiles they share, the higher is the weight that is assigned to them and the more likely they are to be topologically related.

The following schemes are defined:

1) *Co-occurrence Frequency* (CF) simply counts the tiles shared by s and t, i.e., $CF(s,t) = |B_s \cap B_t|$, where B_k stands for the set of tiles/blocks containing geometry k.

2) *Jaccard Similarity* (JS) normalizes the overlap similarity defined by CF, i.e., $JS(s,t) = \frac{|B_s \cap B_t|}{|B_s|+|B_t|-|B_s \cap B_t|}$, capturing the idea that the portion of tiles shared by two geometries is proportional to the likelihood that they satisfy a positive topological relation.

3) *Pearson's χ^2 test*, which is inspired from the Entity Resolution approach BLAST [21], extends CF by assessing whether two geometries s and t appear independently in the set of tiles. To infer their dependency, it estimates whether the distribution of tiles containing s in B is the same as the distribution if we exclude the tiles that contain t. In more detail, it uses the *contingency table* (see Table 1), where $n_{1,1}$ stands for the number of tiles shared by the two geometries, $n_{1,2}$ for the number of tiles containing s but not t, $n_{2,1}$ for the number of tiles containing t but not s, and $n_{2,2}$ for the number of tiles containing neither geometry. These are the observed values, whereas the expected value for each cell of the contingency table is $m_{i,j} = \frac{n_{i,+} \cdot n_{+,j}}{n_{+,+}}$. In this context, each pair of geometries s and t is weighted according to the following formula: $w_{i,j} = \sum_{i \in \{1,2\}} \sum_{j \in \{1,2\}} \frac{n_{i,j} - m_{i,j}}{m_{i,j}}$.

	t	¬t	
s	$n_{1,1}$	$n_{1,2}$	$n_{1,+}$
¬s	$n_{2,1}$	$n_{2,2}$	$n_{2,+}$
	$n_{+,1}$	$n_{+,2}$	$n_{+,+}$

Table 1: Contingency table for geometries s and t.

Note that CF is the only weighting scheme that relies on *local information*, i.e., information that pertains exclusively to the pair of geometries at hand. For this information, it suffices to index the source dataset so that we know the source geometries in the tiles that contain a particular target geometry. In contrast, JS and χ^2 require the total number of tiles as well as the number of tiles per geometry. This *global information* can only be computed by indexing both datasets, which increases the run-time and complicates the weight estimation in the context of the MapReduce framework.

To overcome this drawback, we consider their approximations, which replace the actual number of tiles per geometry with the maximum one: they count the tiles intersecting the MBR of a target (source) geometry, independently of the existence of source (target) geometries. For instance, assume that $S = \{g_1, g_2, g_4\}$ and $T = \{g_3\}$ in Figure 1: the MBR of g_3 intersects 9 tiles, but only 6 of them contain a source geometry; the tiles/blocks b_{03}, b_{13} and b_{23} contain no source geometry and, thus, are disregarded by the original definitions of JS and χ^2. They are considered, though, by their approximations, which produce more noisy weights, but save the time and space required to index the target dataset.

6 PROGRESSIVE ALGORITHMS

The following progressive methods operate in three steps: (i) Filtering is applied to reduce the computational cost to geometries with MBR(s) located in the same tile(s); (ii) *Scheduling* defines the processing order of the candidate pairs produced by Filtering, without verifying any of them. It tries to maximizes Progressive Geometry Recall, based on heuristics; (iii) Verification examines the candidate pairs in the order specified by Scheduling.

6.1 Progressive RADON

This approach uses RADON to index both input datasets, yielding a large set of fine-grained tiles. Then, it processes these tiles in decreasing or increasing size, i.e., total number of geometries. As we explain in Section 8.2, the best ordering depends on the data at hand. Inside every tile, all pairs of geometries are sorted in decreasing weight, as it is defined by one of the above weighting schemes. Thus, the pairs that are most likely to have a non-trivial topological

Algorithm 2: Progressive RADON

```
   input  : the source & target datasets, S and T, resp., the set of
            non-trivial topological relations R, the budget BU, the
            pair weighting scheme W and the tile ordering scheme O
   output : the links L = {(s,r,t) | s ∈ S ∧ t ∈ T ∧ r ∈ R ∧ r(s,t)}
   /* Filtering step                                              */
1  I ← {} ;                          // Equigrid index structure
2  (Δx, Δy) ← defineIndexGranularity(S, T);
3  ... ;              /* Index S as in Alg. 1, Lines 1-12 */
   /* Index target dataset                                        */
13 foreach geometry t ∈ T do
14     (x1(t), y1(t), x2(t), y2(t)) ← getDiagCorners(t);
15     for i ← ⌊x1(t) · Δx⌋ to ⌈x2(t) · Δx⌉ do
16         for j ← ⌊y1(t) · Δy⌋ to ⌈y2(t) · Δy⌉ do
17             I.addToIndex(i, j, t);
18             j ← j + 1;
19         end
20         i ← i + 1;
21     end
22 end
   /* Scheduling step                                             */
23 L ← {}; verifications = 0;
24 B ← I.getTiles();
25 B′ ← O.orderBySize(B);
26 E ← getGeometryIndex(B′);
   /* Verification step                                           */
27 foreach tile b_i ∈ B′ do
28     C ← {} ;                  // The set of candidate pairs
29     foreach pair < s, t > ∈ b_i do
30         if intersectingMBRs(s, t) & refPoint(b_i, s, t) then
31             w_{s,t} ← W.getWeight(E, s, t);
32             C ← C ∪ {{s, t}, w_{s,t}};
33         end
34     end
35     C′ ← orderInDecreasingWeight(C);
36     foreach pair < s, t > ∈ C′ do
37         verifications ← verifications +1;
38         if verifications == BU then
39             return L;
40         end
41         IM ← verify(s, t);
42         L ← L ∪ IM.getRelations();
43     end
44 end
45 return L;
```

relation are processed first inside every tile. A *geometry index* is created to associate every geometry id with the ids of the tiles that contain it (see Figure 3). This is necessary for estimating the weights of the hit probability schemes inside every tile.

To avoid redundant verifications across different tiles, we replace RADON's hashmap with the *reference point technique* [5]: for each pair of candidates, the verification is carried out only in the tile that

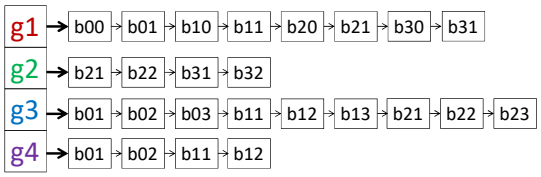

Figure 3: The geometry index of the geometries in Figure 1.

contains the top-left corner of the intersection of their MBRs. As an example, see Figure 1, where the geometries g_1 and g_2 are only verified in tile b_{21}, which contains the top-left corner of their intersection. The entire processing terminates as soon as the number of verified pairs exceeds the predetermined budget BU.

This method is outlined in Algorithm 2. Lines 1-12 add the source geometries to the EquiGrid index, just as the Lines 1-12 of Algorithm 1, except that its dimensions are defined by considering both input datasets: $\Delta_x = 1/2 \cdot (mean_{s \in S} MBR(s).width + mean_{t \in T} MBR(t).width)$ and $\Delta_y = 1/2 \cdot (mean_{s \in S} MBR(s).length + mean_{t \in T} MBR(t).length)$. The target geometries are indexed in Lines 13-22. So far, there is no difference from RADON's batch functionality, i.e., we apply the same Filtering approach. The progressive processing starts from the Scheduling step in Line 23: the set of tiles/blocks B is retrieved from the EquiGrid index (Line 24) and is sorted by (decreasing or increasing) size (Line 25). In Line 26, the geometry index E is created, associating every geometry id with the ids of the tiles that contain it. Subsequently, the Verification step processes every tile b_i as follows (Lines 27-44): each pair of geometries with intersecting MBRs and their reference point in b_i (Line 30) is weighted according to the given scheme W (Line 31) and added to the local set of candidates (Line 32). Next, all candidate pairs in the current tile are sorted and verified in decreasing weight, until the total number of verifications reaches BU (Lines 35-43).

The space complexity of this approach is $O(|S|+|T|+|B|)$, where the last term represents the memory required for maintaining in memory both the set of tiles and the geometry index. The time complexity for the Filtering step is $O(|S|+|T|)$, as it considers both input datasets, while for the Scheduling step it is $O(|B| \cdot \log |B|)$, which corresponds to the sorting of all tiles. Finally, the Verification step is dominated by the examination of $|BU|$ pairs, i.e., $O(|BU|)$.

6.2 Progressive GIA.nt

To turn GIA.nt into a progressive approach, we maintain a min-max priority queue with the BU most promising pairs of geometries from the entire input datasets. To populate this queue, progressive GIA.nt begins with indexing only the source dataset. Then, it reads the target geometries from the disk one by one and for each of them, it gathers all distinct source geometries in the tiles that intersect its MBR. Every pair of geometries $< s, t >$ is weighted according to the selected weighting scheme. If its weight is higher than the current minimum weight of the queue, $< s, t >$ is added to the queue. Whenever the size of the queue exceeds BU, the pair with the lowest weight is evicted. In the end, the queue contains the BU top weighted pairs of the entire input datasets. Thus, Progressive GIA.nt produces a global ordering of pairs, which should outperform the local, tile-level ordering that lies at the core of Progressive RADON.

The details of Progressive GIA.nt are outlined in Algorithm 3. Lines 1-12 applies Filtering, which indexes the source dataset in

Algorithm 3: Progressive GIA.nt

```
   input  : the source dataset S, a reader for the target one rd(T),
            the set of non-trivial topological relations R, the budget
            BU & the weighting scheme W
   output : the links L = {(s,r,t) | s ∈ S ∧ t ∈ T ∧ r ∈ R ∧ r(s,t)}
   /* Filtering step                                              */
 1 I ← {} ;                          // Equigrid index structure
 2 (Δx, Δy) ← defineIndexGranularity(S);
 3 ... ;                 /* Index S as in Alg. 1, Lines 1-12 */
   /* Scheduling step                                             */
13 minWeight = 0.0 ; TC ← {} ;                 // Priority queue
14 flags[] ← {}; frequency[] ← {};
15 while rd(T).hasNext() do
16     tm ← rd(T).next() ;          // The current target geom.
17     CS ← {} ;          // The set of candidate source geom.
18     (x1(tm), y1(tm), x2(tm), y2(tm)) ← getDiagC(tm);
19     for i ← ⌊x1(tm) · Δx⌋ to ⌈x2(tm) · Δx⌉ do
20         for j ← ⌊y1(tm) · Δy⌋ to ⌈y2(tm) · Δy⌉ do
21             foreach sn ∈ I.getTileContents(i, j) do
22                 if flags[n] ≠ m then
23                     flags[n] = m ;
24                     frequency[n] = 0;
25                     CS ← CS ∪ sn;
26                 end
27                 frequency[n]++;
28             end
29             j ← j + 1;
30         end
31         i ← i + 1;
32     end
33     foreach geometry sn ∈ CS do
34         if intersectingMBRs(s, t) then
35             ws,t ← weight(frequency[n], sn, tm);
36             if minWeight < ws,t then
37                 TC.add({s, t}, ws,t);
38                 if BU < TC.size() then
39                     head = TC.pop();
40                     minWeight = head.getWeight();
41                 end
42             end
43         end
44     end
45 end
   /* Verification step                                           */
46 L ← {} ;                       // The set of detected links
47 while Tm ≠ {} do
48     tail = Tm.popLast();
49     IM ← verify(tail.s, tail.t);
50     L ← L ∪ IM.getRelations();
51 end
52 return L;
```

the same way as GIA.nt. Next, the Scheduling step creates three data structures: (i) T_C (Line 13) is a min-max priority queue that stores the top-BU weighted pairs in decreasing order of weight. As a result, its head, which is retrieved by $pop()$, always contains the pair with the minimum weight, while the pair with maximum weight is located at its tail, which is retrieved by $popLast()$. (ii) the `int` array $flags$ (Line 14) designates the id of the target geometry that was last associated with a specific source geometry. (iii) the `int` array $frequency$ (Line 14) counts the number of occurrences of a specific source geometry in the tiles associated with the current target geometry. In essence, it measures the number of tiles shared by the two geometries. E.g., $frequency[1] = 5$ and $flags[1] = 2$ mean that the 2^{nd} source geometry (id=1) shares 5 tiles with the 3^{rd} target geometry (id=2). Thus, these two arrays facilitate the computations of the hit probability schemes.

The three data structures are populated by the loop in Lines 15-45. Lines 19-21 identify the tiles that should contain the current target geometry t_m, where m denotes its id. For each source geometry in these tiles s_n, where n denotes its id, we check whether it has already appeared in another tile of t_m. If not (Line 22), the arrays $flags$ and $frequency$ are updated accordingly (Lines 23-24) and s_n is added in the set of candidate related geometries C_S (Line 25). Next, its frequency of tile co-occurrence with t_m is incremented (Line 27). The distinct candidate geometries are then weighted based on their tile co-occurrence frequency (Line 35) as long as their MBRs intersect that of t_m (Line 34). The weighted pairs are then added to the priority queue T_C (Line 37) if their weight exceeds the minimum one (Line 36). If T_C contains more pairs than the input budget, its head is removed and the minimum weight threshold is updated accordingly (Lines 38-41). Finally, the Verification step examines the pairs in the priority queue in decreasing order of weight, iteratively retrieving its tail (Lines 47-48). In each case, the relations that are extracted from the intersection matrix IM are added to the list of links L, which is returned as output (Lines 49-50).

The space complexity of Progressive GIA.nt is $O(|S| + |BU|)$, because it suffices to maintain in memory the smallest input dataset, while its tiles involve only source geometries. Its time complexity amounts to $O(|S|)$ for the Filtering and to $O(|T| \cdot |\bar{C_S}| \log |BU|)$ for the Scheduling step, where $|\bar{C_S}|$ is the average number of candidate source geometries per target geometry and $\log |BU|$ is the worst-case cost of inserting a candidate pair in the priority queue. For the Verification step, its time complexity is $O(|BU|)$. We should stress, though, that Progressive GIA.nt is more time efficient than Progressive RADON with respect to weight estimations: the co-occurrence frequency that lies at the core of our weighting schemes can be efficiently estimated by Progressive GIA.nt while gathering the distinct co-occurring source geometries per target geometry. Instead, Progressive RADON weights every pair by comparing the tile ids associated with every geometry in the Geometry Index, which is a time consuming process.

7 MASSIVE PARALLELIZATION

We now explain how to parallelize (Progressive) GIA.nt according to the MapReduce framework. Our approach, which is outlined in Figure 4, loads both datasets as RDDs that are spatially partitioned, based on GeoSpark's Quad-Tree [28]. The source and target RDDs are partitioned using the same partitioner and thus, the topologically close geometries belong to partitions with the same partition id. The RDDs with the same partition id are then merged such that each partition contains all geometries from both datasets that lie within its area. This way, we ensure that all geometries that are likely to satisfy a topological relation coexist in the same partitions.

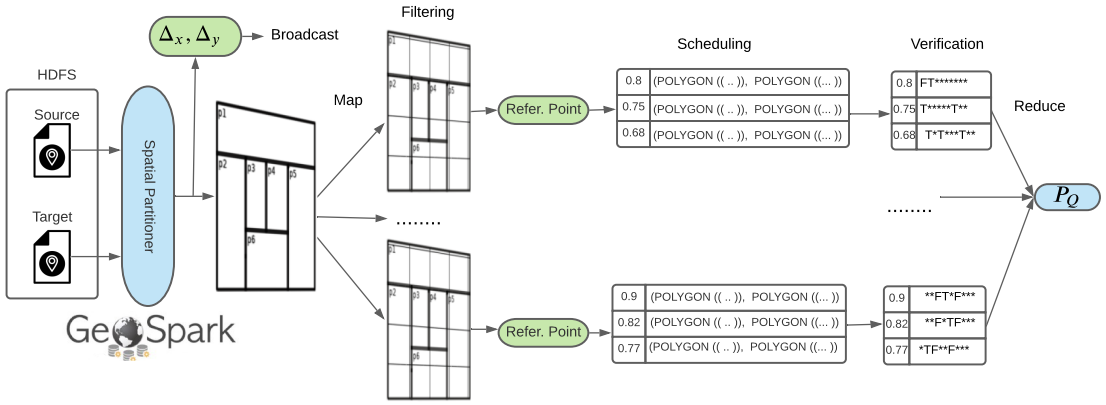

Figure 4: Parallel (Progressive) GIA.nt on top of Apache Spark. Both datasets are loaded in HDFS and spatially partitioned using GeoSpark's Quad-Tree. Map assigns each partition to an Executor, which applies Filtering to find the candidate pairs and the reference point technique to discard the redundant ones. For Progressive GIA.nt, Scheduling orders locally the candidate pairs according to the selected weighting scheme. Verification computes the Intersection Matrices of the selected pairs and Reduce aggregates the qualifying pairs detected by all Executors.

For GIA.nt, each Executor receives one of these partitions as input, during the Map phase. It indexes the source geometries and for each target geometry t, it estimates the tiles that intersect its MBR. Using the index, it retrieves the *distinct* source geometries in these tiles and verifies their topological relations with t. All qualifying pairs are aggregated during the Reduce phase.

The granularity of space tiling is the same as in serial GIA.nt, hence requiring the computation of $\Delta_x = mean_{s \in S} MBR(s).width$ and $\Delta_y = mean_{s \in S} MBR(s).length$ by the Driver, which broadcasts them to the Executors. By caching the source RDD after loading it, we avoid the re-execution of the execution plan so that there is no impact on the performance of the algorithm.

For Progressive GIA.nt, every Executor receives as input a partition of both input datasets, during the Map phase, and applies Filtering to index the source geometries. Then, it processes the target geometries one by one, estimating their weights with the intersecting source geometries. Next, the Executor verifies the top-k weighted pairs, where k is the local budget that is derived by dividing the global budget BU among the data partitions in proportion to the source geometries they contain – the target geometries are not taken into account, as they are not bulk loaded beforehand, but are read on-the-fly, one by one, similar to the serial implementation of (Progressive) GIA.nt. The qualifying pairs of each Executor are aggregated by the Reduce phase.

Note that no data shuffling is required during Scheduling for the weight estimations, since all necessary information is locally available: every Executor estimates all tiles that should contain every geometry. Thus, each Executor operates independently of the others. Note also that the global ordering of the serial implementation is approximated by the local ones in each Executor in order to promote concurrency, making the most of massive parallelization.

We should also stress that both algorithms employ the reference point technique [5] to avoid redundant pairs. This is because every geometry that crosses the borders between two partitions is added to both of them during spatial partitioning. To avoid the resulting redundancy, both algorithms ensure that every pair is verified only in the partition that contains the top left corner of their intersection.

Finally, it is worth noting that spatial partitioning yields uneven partitions, which are skewed with respect to the volume of data and the corresponding computational cost. That is, some partitions are overloaded and require significant time, while others complete their jobs instantaneously, leaving the corresponding nodes idle. To tackle this issue, both algorithms take special care of the overloaded partitions, whose size exceeds significantly the average size of all partitions. After completing the processing of the well-balanced partitions, the entities of the overloaded partitions are indexed and re-partitioned using a HashPartitioner that is based on tiles id. In this way, geometries indexed in the same tiles will be placed in same partitions, thus missing no candidate pairs. Redundant pairs are again discarded with the reference point technique. This is an effective and efficient strategy as long as it applies to a small portion of the input data, because it requires the duplication of each entity as many times as the numbers its tiles.

8 EXPERIMENTAL ANALYSIS

We now present the experiments we performed in order to compare our algorithms to RADON, the current state-of-the-art [19, 20], in terms of effectiveness, time efficiency and memory footprint.

Experimental Setup. All serial methods and experiments were implemented in Java 8. The experiments were ran on a server with Intel Xeon E5-4603 v2 @ 2.2GHz, 128 GB RAM, running Ubuntu 14.04.5 LTS. For RADON's implementation, we used LIMES version 1.7.1[12]. For all time measurements, we used a single physical core and performed three repetitions, reporting the average.

All parallel methods and experiments were implemented in Scala 2.12 using Spark 2.4.4. The experiments were performed on a cluster that runs the Hopsworks data platform [9]. The main module of Hopsworks is Hops[13], which is a next generation distribution of Apache Hadoop, using a new implementation of HDFS called HopsFS [15]. For the experiments we used 30 Executors with 2 cores each, and 10GB of memory.

Our code is available at: https://github.com/giantInterlinking/prGIAnt.

Datasets. The technical characteristics of the real datasets we use in our experiments are reported in Table 2. All of them have

[12]https://github.com/dice-group/LIMES
[13]https://github.com/hopshadoop/hops

	D_1	D_2	D_3	D_4	D_5	D_6
Source Dataset	AREAWATER	AREAWATER	Lakes	Parks	ROADS	Roads
Target Dataset	LINEARWATER	ROADS	Parks	Roads	EDGES	Buildings
#Source Geometries	2,292,766	2,292,766	8,419,320	9,961,891	19,592,688	72,339,926
#Target Geometries	5,838,339	19,592,688	9,961,891	72,339,926	70,380,191	114,796,567
Cartesian Product	$1.34 \cdot 10^{13}$	$4.49 \cdot 10^{13}$	$8.39 \cdot 10^{13}$	$7.21 \cdot 10^{14}$	$1.38 \cdot 10^{15}$	$8.30 \cdot 10^{15}$
#Qualifying Pairs	2,401,396	199,122	5,551,014	14,163,325	163,982,135	1,041,562
`#Contains`	806,158	3,792	947,788	6,323,433	12,218,867	276,010
`#CoveredBy`	0	0	3,031,403	48,922	53,758,452	83,936
`#Covers`	832,843	4,692	948,086	6,470,655	12,218,867	276,023
`#Crosses`	40,489	106,823	270,248	6,490,937	6,769	314,708
`#Equals`	0	0	557,465	3,147	12,218,867	18,972
`#Intersects`	2,401,396	199,122	5,551,014	14,163,325	163,982,135	1,041,562
`#Overlaps`	0	0	822,241	45,116	73	54,899
`#Touches`	1,554,749	88,507	1,037,412	1,258,163	110,216,841	332,249
`#Within`	0	0	3,030,790	48,823	53,758,452	82,668
Total Topological Relations	5,635,635	402,936	16,196,447	34,852,521	418,379,323	2,481,027

Table 2: Technical characteristics of the real datasets for Geospatial Interlinking.

been widely used in the literature [7, 25] and are publicly available (http://spatialhadoop.cs.umn.edu/datasets.html) They contain public data about area hydrography (AREAWATER), linear hydrography (LINEARWATER), roads (ROADS) and all edges (EDGES) in USA. The also contain the boundaries of all lakes (Lakes), parks or green areas (Parks), roads and streets (Roads) as well as of all buildings (Buildings) around the world. Each column of Table 2 shows statistics for a pair (D_1–D_6) of interlinked datasets.

8.1 Batch Geospatial Interlinking

Table 3 compares the performance of GIA.nt and RADON with respect to filtering time (t_f), verification time (t_v) and memory footprint (m) over D_1 and D_2, which are the only dataset pairs that RADON can process with the available memory resources. For brevity, we omit GIA.nt's performance on D_3, D_4 and D_6, and present its performance only on D_5, which is representative of a voluminous dataset pair.

We observe that for both algorithms, Verification is the bottleneck, with Filtering accounting for a negligible portion of the overall run-time. Nevertheless, Filtering manages to reduce the tens of *trillions* pairs considered by the brute-force approach to tens of *millions* candidate pairs, as shown in Table 4. Hence, Filtering is an indispensable step in Geospatial Interlinking.

We also observe that GIA.nt outperforms RADON with respect to t_f by >50%, on average, because its Filtering indexes only the source dataset – unlike RADON, which indexes both input datasets. As shown in Table 4, GIA.nt yields a space tiling of higher dimensionality than RADON, since it defines many more tiles on each axis (#Dimensionality) and overall (#Total Tiles). These tiles involve significantly more geometry pairs than RADON, but the number of non-redundant pairs is much lower. This means that GIA.nt's Filtering reduces the set of candidate pairs, while increasing the co-occurrence patterns of the qualifying pairs, thus boosting the performance of weighting schemes and progressive methods.

During Verification, both algorithms examine the same number of pairs with intersecting MBRs. Yet, GIA.nt's t_v is lower than RADON by > 80%, on average, as the latter approach repeats the entire processing for each topological relation independently of the others – even though most operations, are common. Instead, GIA.nt computes the Intersection Matrix once for each pair of candidates, extracting all their topological relations, as dictated by Definition 1. Seemingly, this improvement is trivial, but in practice RADON cannot be adapted to Definition 1 in a straightforward way, as it involves different filters for each topological relation. Moreover, there are scientific competitions that measure the run-time of Geospatial Interlinking approaches with respect to individual topological relations, such as the Ontology Alignment Evaluation Initiative[14]. Typically, though, real applications do not focus on a specific topological relation, but require all of them in order to produce results of higher quality. For this reason, it is crucial to redefine Geospatial Interlinking as in Definition 1.

	D_1			D_2			D_5		
	t_f (sec)	t_v (min)	m (GB)	t_f (sec)	t_v (min)	m (GB)	t_f (sec)	t_v (hrs)	m (GB)
RADON	56	445.8	64	240	967.0	78	-	-	-
GIA.nt	42	76.7	22	42	166.4	22	98	18.1	85

Table 3: The filtering time (t_f), the verification time (t_v) and the memory footprint (m) of RADON and GIA.nt. For GIA.nt, t_v includes the time required for reading the target dataset from the disk (4.9, 6.4 and 20.1 min for D_1, D_2 and D_5, resp.).

Looking into RADON's run-time per relation in Table 5, we observe that some individual relations are faster than GIA.nt, while others are slower. For the former, RADON involves specialized filters that reduce significantly the number of verified pairs. E.g., for `Equals`, RADON requires the two geometries to have identical MBRs. Given that no geometry pair in our datasets satisfies `Equals` in D_1 and D_2 (see Table 2), there are no identical MBRs among the candidates that are produced by space tiling. As a result, RADON's run-time is extremely low, measuring only the cost of space tiling and the comparison of MBRs among the resulting candidates. In contrast, GIA.nt applies a single, generic filter for all relations: it

[14] https://project-hobbit.eu/challenges/om2020

	D_1	D_2
#Dimensionality ($X \times Y$)	68,493 × 20,999	105,656 × 32,580
#Total Tiles	$1.44 \cdot 10^9$	$3.44 \cdot 10^9$
#Populated Tiles	$4.72 \cdot 10^7$	$1.51 \cdot 10^8$
#Pairs in Tiles	$9.69 \cdot 10^7$	$2.40 \cdot 10^8$
#Unique Pairs	$2.34 \cdot 10^7$	$3.35 \cdot 10^7$

(a) RADON

#Dimensionality ($X \times Y$)	138,325 × 43,559	138,325 × 43,559
#Total Tiles	$6.03 \cdot 10^9$	$6.03 \cdot 10^9$
#Populated Tiles	$5.34 \cdot 10^7$	$4.76 \cdot 10^7$
#Pairs in Tiles	$1.49 \cdot 10^8$	$3.05 \cdot 10^8$
#Unique Pairs	$1.23 \cdot 10^7$	$2.80 \cdot 10^7$

(b) GIA.nt

Table 4: Technical characteristics of the Equigrid indices constructed by RADON's and GIA.nt's Filtering.

	D_1 (min)	D_2 (min)
`Contains`	62.7	139.1
`CoveredBy`	2.0	4.2
`Covers`	61.8	136.7
`Crosses`	81.3	168.5
`Equals`	1.1	3.9
`Intersects`	78.1	174.5
`Overlaps`	78.8	168.8
`Touches`	78.7	168.6
`Within`	1.2	2.7

Table 5: RADON's verification time per topological relation.

merely ensures that the MBRs of the candidates are intersecting, which is a core requirement for all non-trivial topological relations.

The second category of relations includes `Crosses`, `Intersects`, `Overlaps` and `Touches`. For these relations, RADON is slower than GIA.nt, because it merely uses the generic filter of intersecting MBRs, too. GIA.nt is faster by 2.66%, on average, even though its verification time includes the time required for retrieving the target geometries from the disk, unlike RADON. This overhead accounts for 6.39% and 3.85% of GIA.nt's t_v over D_1 and D_2, respectively.

Regarding the memory footprint, we observe that GIA.nt consistently occupies at least 66% less main memory than RADON. This should be attributed not only to the fact that GIA.nt avoids loading the target datasets in main memory, but also to implementation improvements: GIA.nt handles every geometry through its id (rather than its URI, as in RADON), while using data structures that operate on top of primitive data types rather than objects (e.g., `int` instead of `Integer`), based on the GNU Trove library[15].

Finally, Table 3 suggests that as the input size increases from 8 to 22 and 90 million geometries, GIA.nt's run-time increases linearly: from ~1.5 to ~3 and ~18 hours, respectively. It's memory footprint, though, scales sublinearly, from 22GB for D_1 and D_2 to 85GB for D_5, even if we exclusively consider the increase in the size of the source dataset (from 2.3 to 19.6 million geometries). Consequently, GIA.nt's serial execution scales well to very large datasets.

This is verified in Figure 5, which depicts the total run-time for GIA.nt's parallel implementation across all dataset pairs. We observe that GIA.nt processes the largest pair in less than half an hour (26 minutes), while exhibiting a sublinear scalability trend, as

[15] http://trove4j.sourceforge.net/html/overview.html

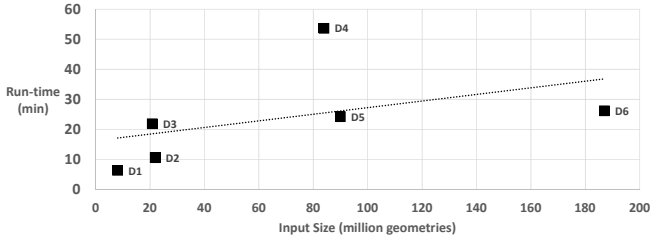

Figure 5: Parallel GIA.nt's run-time across all dataset pairs.

indicated by the fitted linear regression line. The only outlier is D_4, which requires double time than the expected one (53 minutes), because it contains complex geometry collections instead of individual geometries, thus requiring a time-consuming verification.

Discussion. We can conclude that GIA.nt improves RADON in all respects. The verification time and memory footprint are significantly improved, while Filtering yields stronger co-occurrence patterns between qualifying pairs, thus facilitating progressive methods. GIA.nt is also amenable to massive parallelization, unlike RADON. However, Parallel GIA.nt assumes that every pair of geometries involves more or less the same verification cost. This assumption holds in all datasets but D_4, which involves geometry collections. These entities comprise multiple individual geometries, thus raising the verification time of a pair significantly; the larger the geometry collection, the higher the verification cost, leading to poor utilization of resources and higher wall-clock running times.

8.2 Progressive Geospatial Interlinking

To evaluate the progressive methods, we measure their effectiveness in terms of Progressive Geometry Recall (*PGR* - see Section 3.1), Recall=P_Q^D/P_Q^{BU} and Precision=P_Q^D/BU, where P_Q^D stands for the number of detected qualifying pairs and P_Q^{BU} for the number of qualifying pairs *in the given budget BU* (i.e., maximum number of permitted verifications). For all measures, higher values indicate higher effectiveness. For time efficiency, we consider the scheduling and the verification time, t_s and t_v, respectively, disregarding the filtering time, t_f, which is already reported in Section 8.1.

As baseline methods we consider the Optimal approach, which verifies all qualifying pairs before the non-qualifying candidate ones, and the batch algorithms RADON and GIA.nt. For RADON, we exclusively consider its performance with respect to the most generic relation, namely `intersects`. Note that neither RADON nor GIA.nt specify a deterministic processing order for the input data. Instead, their output rate is random, depending on the order the input data. For this reason, we assess their effectiveness within a specific budget by using 100 random permutations of their results and considering the average value for *PGR*, Recall and Precision.

Figure 6 reports the performance of all methods over D_1, D_2 and D_5 for two different budgets: 5 and 10 million verifications. Note that RADON and Progressive RADON cannot process D_5, due to their high space requirements. Note also that for D_1, the number of candidate pairs is ~6.3 million, thus being lower than BU=10M. For this reason, all approaches in Table 6(b) achieve perfect recall and precision over D_1.

For Progressive GIA.nt, we observe that it consistently achieves its best performance in combination with the Jaccard weighting

		Optimal	RADON*	GIA.nt	Pr. RADON (*Dec*)			Pr. RADON (*Inc*)			Progressive GIA.nt		
					CF	*JS*	χ^2	*CF*	*JS*	χ^2	*CF*	*JS*	χ^2
D_1	PGR	0.760	0.396	0.396	0.403	0.403	0.403	0.392	0.392	0.392	0.299	**0.647**	0.641
	Recall	1.000	0.792	0.792	0.799	0.799	0.799	0.796	0.796	0.796	0.726	0.949	0.946
	Precision	0.480	0.381	0.381	0.384	0.384	0.384	0.382	0.382	0.382	0.348	0.456	0.454
	t_s (min)	-	-	-	0.6	0.5	0.6	0.6	0.6	0.6	5.7	5.5	5.3
	t_v (min)	-	61.8	60.6	60.9	57.0	56.2	70.0	69.7	68.3	67.5	**28.4**	28.7
D_2	PGR	0.980	0.159	0.159	0.151	0.151	0.151	0.193	0.193	0.193	0.494	**0.544**	0.507
	Recall	1.000	0.318	0.318	0.297	0.297	0.297	0.366	0.366	0.366	0.646	0.804	0.777
	Precision	0.040	0.013	0.013	0.012	0.012	0.012	0.015	0.015	0.015	0.026	0.032	0.031
	t_s (min)	-	-	-	1.1	1.1	1.0	1.0	1.0	1.1	7.8	7.5	7.6
	t_v (min)	-	53.9	51.6	52.5	52.0	52.3	53.8	53.7	53.7	65.0	**16.2**	16.6
D_5	PGR	0.500	-	0.179	-	-	-	-	-	-	0.121	**0.460**	0.076
	Recall	1.000	-	0.324	-	-	-	-	-	-	0.245	0.926	0.219
	Precision	1.000	-	0.324	-	-	-	-	-	-	0.245	0.926	0.219
	t_s (min)	-	-	-	-	-	-	-	-	-	23.1	22.9	23.5
	t_v (min)	-	-	13.7	-	-	-	-	-	-	29.3	**2.5**	19.8

(a) *BU*=**5,000,000 verifications**

		Optimal	RADON*	GIA.nt	Pr. RADON (*Dec*)			Pr. RADON (*Inc*)			Progressive GIA.nt		
					CF	*JS*	χ^2	*CF*	*JS*	χ^2	*CF*	*JS*	χ^2
D_1	PGR	0.810	0.499	0.500	0.504	0.504	0.504	0.499	0.499	0.499	0.412	**0.694**	0.689
	Recall	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Precision	0.381	0.381	0.381	0.381	0.381	0.381	0.381	0.381	0.381	0.381	0.381	0.381
	t_s (min)	-	-	-	0.6	0.5	0.6	0.6	0.6	0.6	5.7	5.5	5.3
	t_v (min)	-	78.1	76.4	78.4	78.2	78.8	78.8	78.9	78.7	76.6	76.4	76.5
D_2	PGR	0.990	0.318	0.318	0.298	0.298	0.298	0.360	0.360	0.360	0.576	**0.686**	0.665
	Recall	1.000	0.636	0.636	0.589	0.589	0.589	0.664	0.664	0.664	0.821	0.946	0.936
	Precision	0.020	0.013	0.013	0.012	0.012	0.012	0.013	0.013	0.013	0.016	0.019	0.019
	t_s (min)	-	-	-	1.0	1.0	1.1	1.0	1.1	1.1	7.8	7.5	7.6
	t_v (min)	-	105.2	100.9	108.3	107.3	105.5	109.2	108.5	110.1	115.6	**59.9**	60.3
D_5	PGR	0.500	-	0.166	-	-	-	-	-	-	0.121	**0.460**	0.133
	Recall	1.000	-	0.319	-	-	-	-	-	-	0.252	0.929	0.363
	Precision	1.000	-	0.319	-	-	-	-	-	-	0.252	0.929	0.36
	t_s (min)	-	-	-	-	-	-	-	-	-	23.1	22.9	23.5
	t_v (min)	-	-	25.3	-	-	-	-	-	-	52.8	**4.8**	30.1

(b) *BU*=**10,000,000 verifications**

Figure 6: Performance of Progressive RADON and GIA.nt for all weighting schemes in comparison to their batch counterparts and the optimal approach for budgets of 5M and 10M verifications. For RADON, we consider only the relation `intersects`.

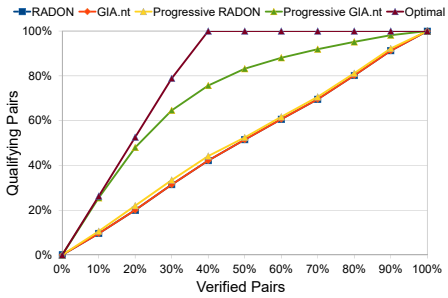

Figure 7: PGR of all approaches over D_1.

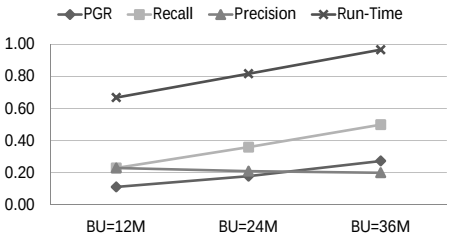

Figure 8: Parallel Progressive GIA.nt over D_4 for 12M, 24M and 36M verifications.

scheme (*JS*). This applies to all effectiveness measures and t_v, suggesting that pairs with high ratio of common tiles are much more similar and, thus, much faster to verify. In contrast, the Co-occurrence Frequency weighting scheme (*CF*) exhibits by far the worst performance in most cases, both with respect to effectiveness and t_v. This means that pairs with many common tiles involve a more time-consuming verification, probably because they correspond to large, complex geometries. The Pearson's χ^2 test lies in the middle of *JS* and *CF* with respect to all measures, except t_s.

In fact, the scheduling time is relatively stable across all weighting schemes for each dataset. The reason is that t_s is dominated by the time required to read the target dataset from the disk (see Table 3 for the actual reading times). Hence, only a small portion of t_s pertains to the cost of weight estimation. Given that each scheme computes weights for all candidate pairs, only the insertions in the priority queue differ between them (e.g., due to ties with the minimum weight), thus yielding minor fluctuations. For the same reasons, t_s is relatively independent of the given budget.

For Progressive RADON, we observe that there is practically no difference in the performance of the three weighting schemes with respect to any effectiveness and time efficiency measure. This should be attributed to two reasons: (i) RADON's filtering yields weak co-occurrence patterns among the qualifying pairs, as explained above, and (ii) its scheduling is dominated by the ordering of tiles. That is, there are significant differences only between the decreasing and the increasing ordering of tiles. Yet, there is no clear winner among them. In D_1, the decreasing ordering is much faster for *BU*=5M and achieves slightly higher effectiveness, while for *BU*=10M, it achieves higher *PGR*, even though all other measures are identical, due to the larger budget. For D_2, though, the increasing order exhibits significantly higher effectiveness than the decreasing one at the cost of slower verification. This should be expected, as Progressive RADON's t_v includes the weighting of candidate pairs and larger tiles involve more pairs to be weighted.

Regardless of its configuration, Progressive RADON significantly underperforms Progressive GIA.nt in terms of effectiveness, except for *CF* weighting over D_1. Regarding time efficiency, t_s is significantly higher for the latter. The reason is that Progressive GIA.nt's Scheduling reads the target dataset from the disk and weights all candidate pairs, unlike Progressive RADON's Scheduling, which merely orders the set of tiles according to their size and differs pair weighting to Verification. The verification time, though, is consistently lower for Progressive GIA.nt in combination with *JS* and x^2, even by >60%, as both promote pairs with low cost. Regarding memory footprint, the relation of the two methods is similar to that of their batch counterparts, with Progressive GIA.nt processing D_5 with the available memory resources, unlike Progressive RADON.

To compare the progressive algorithms with the batch ones, the latter iteratively select at random the pair to be processed next. Their probability of selecting a qualifying pair is equal to the proportion of qualifying pairs in the candidate ones - if this is high, the batch algorithms achieve high PGR. For these reasons, the two batch methods exhibit practically equivalent performance and are quite competitive over D_1, which involves *a high proportion of qualifying to candidate pairs*. Regardless of the budget, the difference between RADON and Progressive RADON is practically insignificant, while GIA.nt outperforms Progressive GIA.nt in conjunction with the *CF* weighting schemes. The latter applies to D_5 even for the χ^2 weighting scheme, as both budgets are much smaller than the number of existing qualifying pairs. In case of D_2, though, *the ratio*

of qualifying to candidate pairs is very low; thus, the progressive approaches outperform the batch ones to a significant extent. On average, across both budgets, Progressive RADON with increasing tile sizes achieves >17% higher PGR and ∼10% higher recall than RADON; Progressive GIA.nt outperforms GIA.nt with respect to PGR, Recall and Precision by 1.62 times, ∼88% and 83%, respectively, on average, across all budgets and weighting schemes. Note also that Progressive GIA.nt with *JS* weights is consistently very close to the optimal algorithm; the higher the ratio of qualifying to candidate pairs is, the closer are the two approaches, and vice versa.

The relative effectiveness of these methods is illustrated in Figure 7, which shows the evolution of qualifying pairs with respect to the candidate ones. We observe that RADON and GIA.nt lie across the diagonal, as implied by their PGR for BU=10M. Progressive RADON is negligibly higher than the two algorithms, but Progressive GIA.nt encloses a significantly larger area under its curve and is located near to the optimal approach. The difference between progressive and batch methods is more prominent in the case of D_2, due to the lower proportion of qualifying pair in the candidate ones. Due to space shortage, we omit the corresponding diagram.

Finally, we should stress that Progressive GIA.nt with *JS* weights is consistently faster across all budgets and datasets among all considered approaches. Its overall running time (t_f+t_s+t_v) over D_1 and D_2 is 50% lower than that of RADON and GIA.nt, on average, for BU=5M and ∼33% faster over D_2 for BU=10M (for BU=10M over D_1, its run-time is higher than the batch approaches, because all methods verify all candidate pairs, but Progressive GIA.nt additionally involves t_s). Compared to the total time required by RADON's original implementation to compute all topological relations, the improvement is larger than a whole order of magnitude.

We should also stress that high time efficiency is achieved by Parallel Progressive GIA.nt in combination with *JS* weights. Its performance over the most time-consuming dataset pair, D_4, is presented in Figure 8 with respect to *PGR*, Recall, Precision and Relative Run-time (i.e., the overall run-time of the progressive approach normalized by the run-time of the batch approach). We observe that Precision starts from a very high level (23% for BU=12M) and decreases gradually but steadily (20% for for BU=36M), converging to the Precision of batch GIA.nt (14M qualifying pairs/75M candidate pairs ≈ 18%). Recall and PGR increase linearly with the size of the budget, and so does the Relative Run-time. Despite weighting 75M geometry pairs, Parallel GIA.nt manages to detect a large portion of the existing qualifying pairs with just 2/3 of the overall run-time (<36 wall-clock minutes). Similarly high performance is achieved over all dataset pairs, but we omit the details, due to lack of space.

9 CONCLUSIONS

In this paper, we defined Holistic Geospatial Interlinking as the task of simultaneously computing all DE-9IM topological relations between the input geometries. To solve it, we proposed GIA.nt, which significantly reduces the overall run-time and the space requirements of RADON. We also proposed Progressive Geospatial Interlinking as the task of computing as many topological relations as possible within a limited budget in terms of pair verifications. We adapted RADON and GIA.nt to address this new task by producing results in a pay-as-you-go manner. Our experiments demonstrate that both algorithms outperform their batch counterparts, with Progressive GIA.nt being much closer to the optimal processing order. Progressive GIA.nt actually detects almost all qualifying pairs among the input data by reducing RADON's run-time by at least an order of magnitude, while maintaining a very low memory footprint. We also adapted (Progressive) GIA.nt to the MapReduce parallelization, verifying their high scalability to voluminous datasets.

In the future, we will adapt our approaches to detecting approximate and metrically-refined topological relations as in [18].

Acknowledgements. This work was partially funded by the EU H2020 project ExtremeEarth (Grant agreement No. 825258) and by Greek national funds, under the call Research-Create-Innovate (project codes: T1EDK-04810 and T2EDK-02848).

REFERENCES

[1] E. P. F. Chan and J. N. H. Ng. A general and efficient implementation of geometric operators and predicates. In *SSD*, volume 1262, pages 69–93, 1997.
[2] E. Clementini, P. D. Felice, and P. van Oosterom. A small set of formal topological relationships suitable for end-user interaction. In *SSD*, pages 277–295, 1993.
[3] E. Clementini, J. Sharma, and M. J. Egenhofer. Modelling topological spatial relations: Strategies for query processing. *Comput. Graph.*, 18(6):815–822, 1994.
[4] I. F. Cruz, F. P. Antonelli, and C. Stroe. Agreementmaker: Efficient matching for large real-world schemas and ontologies. *PVLDB*, 2(2):1586–1589, 2009.
[5] J. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *ICDE*, pages 535–546, 2000.
[6] M. J. Egenhofer and R. D. Franzosa. Point-set topological spatial relations. *International Journal of Geographical Information System*, 5(2):161–174, 1991.
[7] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363. IEEE Computer Society, 2015.
[8] T. Ioannidis, G. Garbis, K. Kyzirakos, K. Bereta, and M. Koubarakis. Evaluating geospatial RDF stores using geographica 2. *CoRR*, abs/1906.01933, 2019.
[9] M. Ismail, E. Gebremeskel, T. Kakantousis, G. Berthou, and J. Dowling. Hopsworks: Improving user experience and development on hadoop with scalable, strongly consistent metadata. In *ICDCS*, pages 2525–2528, 2017.
[10] E. H. Jacox and H. Samet. Spatial join techniques. *ACM TODS*, 32(1):7, 2007.
[11] K. Janowicz, S. Scheider, T. Pehle, and G. Hart. Geospatial semantics and linked spatiotemporal data - past, present, and future. *Semantic Web*, 3(4):321–332, 2012.
[12] O. Kwon and K. Li. Progressive spatial join for polygon data stream. In *SIGSPATIAL*, pages 389–392, 2011.
[13] A. Mobasheri. A rule-based spatial reasoning approach for openstreetmap data quality enrichment; case study of routing-navigation. *Sensors*, 17(11):2498, 2017.
[14] A. N. Ngomo. ORCHID - reduction-ratio-optimal computation of geo-spatial distances for link discovery. In *ISWC*, pages 395–410, 2013.
[15] S. Niazi, M. Ismail, S. Haridi, and J. Dowling. Hopsfs: Scaling hierarchical file system metadata using newsql databases. In *Enc. of Big Data Technologies*. 2019.
[16] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259–270, 1996.
[17] M. Perry and J. Herring. Ogc geosparql-a geographic query language for rdf data. *OGC implementation standard*, 40, 2012.
[18] B. Regalia, K. Janowicz, and G. McKenzie. Computing and querying strict, approximate, and metrically refined topological relations in linked geographic data. *Trans. GIS*, 23(3):601–619, 2019.
[19] T. Saveta, I. Fundulaki, G. Flouris, and A. N. Ngomo. Spgen : A benchmark generator for spatial link discovery tools. In *ISWC*, pages 408–423, 2018.
[20] M. A. Sherif, K. Dreßler, P. Smeros, and A. N. Ngomo. Radon - rapid discovery of topological relations. In *AAAI*, pages 175–181, 2017.
[21] G. Simonini, S. Bergamaschi, and H. V. Jagadish. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *PVLDB*, 9(12):1173–1184, 2016.
[22] G. Simonini, G. Papadakis, T. Palpanas, and S. Bergamaschi. Schema-agnostic progressive entity resolution. *IEEE TKDE*, 31(6):1208–1221, 2019.
[23] P. Smeros and M. Koubarakis. Discovering spatial and temporal links among RDF data. In *Proceedings of the Workshop on Linked Data on the Web*, 2016.
[24] W. H. Tok, S. Bressan, and M. Lee. Progressive spatial join. In *SSDBM*, pages 353–358, 2006.
[25] D. Tsitsigkos, P. Bouros, N. Mamoulis, and M. Terrovitis. Parallel in-memory evaluation of spatial joins. In *SIGSPATIAL*, pages 516–519, 2019.
[26] S. E. Whang, D. Marmaros, and H. Garcia-Molina. Pay-as-you-go entity resolution. *IEEE TKDE*, 25(5):1111–1124, 2013.
[27] Z. Yin, C. Zhang, D. Goldberg, and S. Prasad. An nlp-based question answering framework for spatio-temporal analysis. In *ICGDA*, pages 61–65, 2019.
[28] J. Yu, Z. Zhang, and M. Sarwat. Spatial data management in apache spark: the geospark perspective and beyond. *GeoInformatica*, 23(1):37–78, 2019.