Static and Dynamic Progressive Geospatial Interlinking

GEORGE PAPADAKIS and GEORGE MANDILARAS, National and Kapodistrian University of Athens, Greece
NIKOS MAMOULIS, University of Ioannina, Greece
MANOLIS KOUBARAKIS, National and Kapodistrian University of Athens, Greece

Geospatial data constitute a considerable part of Semantic Web data, but at the moment, its sources are insufficiently 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 all their geometries. In this work, we address this issue by defining 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 a static progressive algorithm, which employs a fixed processing order, and a dynamic one, whose processing order is updated whenever new topological relations are discovered. We equip both algorithms with a series of weighting schemes and explain how they can be adapted to massive parallelization with Apache Spark. We conduct a thorough experimental study over six large, real datasets, demonstrating the superiority of our techniques over the current state-of-the-art. Special care is also taken to analyze the performance of the various weighting schemes.

CCS Concepts: • **Information systems** → **Information integration**; **Spatial-temporal systems;**

Additional Key Words and Phrases: Geospatial interlinking, DE-9IM relations, progressive processing

ACM Reference format:
George Papadakis, George Mandilaras, Nikos Mamoulis, and Manolis Koubarakis. 2022. Static and Dynamic Progressive Geospatial Interlinking. *ACM Trans. Spatial Algorithms Syst.* 8, 2, Article 16 (April 2022), 41 pages.
https://doi.org/10.1145/3510025

1 INTRODUCTION

The Web abounds in huge datasets of geospatial data, such as OpenStreetMap[1] and the U.S. Census Bureau TIGER files.[2] Alone, Geographica 2.0 [9] has gathered almost half a billion RDF triples from

[1] https://www.openstreetmap.org.
[2] http://spatialhadoop.cs.umn.edu/datasets.html.

This work was partially funded by the EU H2020 project ExtremeEarth (Grant No. 825258) and by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant" (Project No. HFRI-FM17-2351).
Authors' addresses: G. Papadakis, G. Mandilaras, and M. Koubarakis, National and Kapodistrian University of Athens, Greece; emails: {gpapadis, gmandi, koubarak}@di.uoa.gr; N. Mamoulis, University of Ioannina, Greece; email: nikos@cs.uoi.gr.

2374-0353/2022/04-ART16 $15.00
https://doi.org/10.1145/3510025

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 [3, 4, 7]. These relations are indispensable for crucial applications like the Icewatch project,[5] which integrates observational data from ships navigating the arctic with geospatial information from satellite images offered by Earth observation programmes, such as Landsat.[6] Due to the climate change, this information needs to be frequently and rapidly updated to ensure safe ship navigation.

Moreover, applications involving reasoning [13, 16], question answering [34] or simply running GeoSPARQL queries over geospatial data [22] 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[7]: even though it corresponds to almost 20% of the LOD cloud triples, only 7% of the triples linking different datasets pertain to geometries [17].

Such applications can be facilitated by *Geospatial Interlinking* [26, 27, 30], 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.[8] The two graphs are then merged 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 [2]. Therefore, existing Geospatial Interlinking approaches [26, 27, 30] scale to large volumes of data only 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 that co-occur in at least one tile to identify the topological relations they satisfy.

In Reference [19], *GIA.nt* was presented as the state-of-the-art approach for Geospatial Interlinking. Its Filtering step depends exclusively on the source dataset, defining an adaptive Equigrid, where the granularity of both axis is derived from the corresponding average length of the source geometries' MBR. In this way, it avoids loading the target dataset into main memory, unlike other established methods like RADON [27]. Instead, the target geometries are read and verified one by one from the disk, thus reducing the memory consumption by more than 50%.

However, the batch functionality of GIA.nt is not suitable in the context of *sparsity*: in case a mere fraction of the input data is related, most of the processing time is spent on verifying pairs of geometries that have intersecting MBRs, but are topologically disjoint. Most importantly, batch interlinking is not suitable for applications with limited temporal or computational resources.

[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://lod-cloud.net.
[8] https://locationtech.github.io/jts/jts-faq.html.

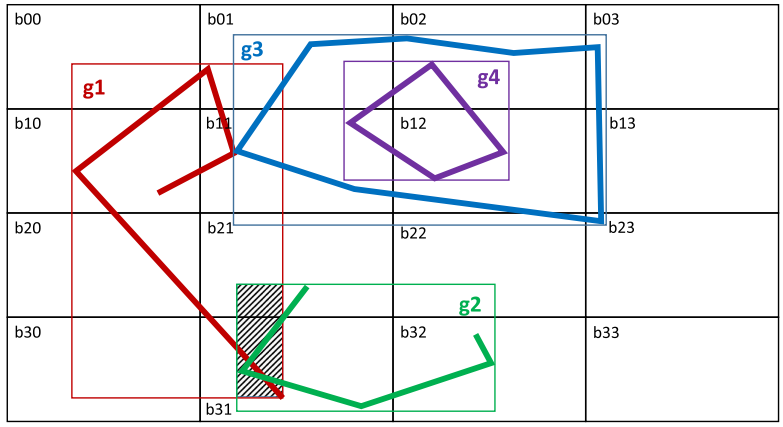

Fig. 1. The space tiling approach for four geometries, where g_1 is a LineString that `intersects` LineString g_2 and `touches` Polygon g_3, which `contains` Polygon g_4. The shaded area corresponds to the intersection of the MBRs of g_1 and g_2.

These are applications that typically run on the cloud, but have a limited budget for exploiting utilities like the AWS Lambda functions,[9] which charge the user whenever they are called.

As a concrete example consider the ice monitoring application presented in Reference [15], which facilitates the safety of ship navigation in the Arctic through Geospatial Interlinking. Its input data comprise in situ observations about icebergs (e.g., their location at specific times and other important properties of sea ice) and satellite images from the EU Copernicus Programme.[10] Its goal is to interlink these large data sources for visualization purposes as well as for performing analytics. Given, though, that the application runs on the HopsWorks platform,[11] which is only available in specific, limited time slots, it cannot rely on a batch interlinking process.

A similar application is developed in the context of the H2020 Extreme Earth research project.[12] This time, it monitors the water provided by the seasonal snow melt, which is essential for growing regions and crop development. By interlinking satellite images from the EU Copernicus Programme with in situ observations and agricultural datasets, it analyzes and provides insights for high precision farming. This application also runs on the HopsWorks platform, thus being incompatible with batch interlinking, too.

To facilitate such applications, *Progressive GIA.nt* [19] goes beyond GIA.nt and all other batch algorithms by maximizing the throughput of Geospatial Interlinking within the available temporal or computational resources. These resources are typically determined as a maximum number of verified pairs of geometries, called ***budget* (*BU*)**. To make the most of them, Progressive GIA.nt interposes a Scheduling step between Filtering and Verification with the goal of weighting all geometry pairs according to their estimated likelihood that they involve related geometry pairs. The top-*BU* weighted ones are placed in a priority queue so that the most promising pair, located at the tail of the queue, is iteratively verified. Ideally, the related pairs are verified before the non-related ones. In practice, though, this depends on the effectiveness of the selected weighting scheme, i.e., on how higher are the scores it assigns to the related pairs than the scores of the non-related ones.

[9] https://aws.amazon.com/lambda.
[10] https://www.copernicus.eu.
[11] https://www.hopsworks.ai.
[12] http://earthanalytics.eu.

In this work, we extend our work in Reference [19] in the following directions:

- We introduce *Dynamic Progressive GIA.nt*, which goes beyond the current *static approach* by updating the processing order of geometry pairs on-the-fly, based on the topological relations that are detected. Its goal is to leverage the evidence provided by the latest verifications to promote pairs that are more likely to be related, too. We also explain in detail how it can be parallelized on top of Apache Spark.
- We introduce two new weighting schemes for progressive geospatial interlinking. One that achieves high effectiveness based on the overlap of the MBRs and one that maximizes time efficiency, based on the complexity of the geometry pairs in terms of the number of points forming their boundaries. Both schemes apply seamlessly to static and dynamic progressive GIA.nt.
- We consider composite weighting schemes as a means of addressing ties to minimize the randomness in the ranking of geometry pairs. They consist of a primary scheme that is replaced by a secondary one of high distinctiveness in case of ties, thus ensuring a more deterministic functionality.
- We extend the experimental analysis in the following ways: (i) we report the detailed performance of all serialized and parallel algorithms on top of all six, real-world datasets, (ii) we examine the relative performance of all weighting schemes based on the characteristics of their top pairs, and (iii) we elaborate on the relative performance of static and dynamic Progressive GIA.nt.

The rest of the article is structured as follows: Section 2 discusses the main works in the field, while Section 3 provides the necessary background knowledge. In Section 4, we present GIA.nt, the state-of-the-art algorithm for batch geospatial interlinking, while its static and dynamic progressive counterparts are coined in Section 5, together with the necessary weighting schemes. Section 6 presents the massive parallelization of all algorithms on top of Apache Spark, Section 7 delves into our experimental analysis, and Section 8 concludes the article along with directions for future works.

We have publicly released the implementation of all serial algorithms at https://github.com/giantInterlinking/prGIAnt. The implementation of all parallel algorithms is publicly available at https://github.com/GiorgosMandi/DS-JedAI.

2 RELATED WORK

The main algorithms for Geospatial Interlinking are *Silk-spatial* [30], *RADON* [27], *RADON2* [1], and *stLD* [24, 25].

For Filtering, Silk-spatial 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, in the sense that they involve a large number of geometry pairs. Thus, too many pairs are verified, incurring a computational cost that is close to that of a brute-force approach [27]. This is partially ameliorated through massive parallelization on top of Apache Hadoop.[13] Its Verification step considers a single topological relation, even though it uses the same Filtering for all relations. This means that the entire algorithm is repeated from scratch if another relation needs to be examined over the same data.

RADON 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 source and target geometry MBRs in that dimension. The Filtering step also involves a swapping strategy, which goes through

[13] https://hadoop.apache.org.

all source and target geometries to identify the dataset with the smallest Estimated Total Hypervolume, in an effort to minimize the size of the Equigrid. As a result, though, RADON needs to maintain both the source and the target datasets in main memory, resulting in very high space requirements. 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 geometries with intersecting MBRs that co-occur in multiple tiles. To avoid verifying such pairs more than once, RADON maintains in main memory a hash-table with all geometry pairs verified so far. Yet, this renders its massive parallelization non-trivial: special care should be taken to partition the input data among the available workers in a way that avoids all redundant verifications (broadcasting all geometries to all workers is not an option for large datasets, due to the high memory requirements). The Verification step operates at the level of individual topological relations, incorporating specialized filters for some of them. For instance, `equals` is verified only after ensuring that the source and target geometries have identical MBRs. Similar to Silk-spatial, though, the entire algorithm is repeated whenever another relation is examined over the same data.

RADON2 extends RADON so that it avoids this problem. Using the same Filtering approach, it alters the Verification step so that it computes simultaneously all topological relations for each pair of candidate geometries. As explained in Section 3.1, it computes the Intersection Matrix and extracts all topological relations with simple logical conditions.

stLD is more similar to GIA.nt, because it exploits massive parallelization (on top of Apache Flink[14]) and loads only the source dataset in main memory, while the target geometries are read one by one on the fly, thus accommodating streaming data. Its Filtering step supports a wide variety of indices, such as R-Trees. Among grid indices, it supports a static Equigrid, similar to Silk-spatial, and a hierarchical grid, which combines multiple Equigrids of different granularities. To reduce the number of candidate pairs, it uses MaskLink, a technique that is suitable when the tiles of the source geometries involve large empty areas. MaskLink computes the cell mask, i.e., the polygon of the empty space left by the source geometries, and compares it with the overlap of a target geometry with the current tile; if the former contains the latter, then no candidate pair is verified in this cell. The Verification step supports both proximity and topological relations. Unlike GIA.nt, though, it performs independent computations for every topological relation.

Note that the generic schema matching system *AgreementMaker* [5] has also been adapted to Geospatial Interlinking, outperforming Silk-spatial with respect to time efficiency [26]. However, the thorough experimental study in Reference [26] demonstrates that RADON is significantly faster than both Silk-spatial and AgreementMaker.

Note also that Geospatial Interlinking is relevant to traditional spatial joins, which mainly look for intersecting geometry pairs or pairs of nearby points. Geospatial Interlinking examines a variety of topological relations, but any pair of related geometries satisfies at least the intersection relation, as explained in Section 3. Partition-based spatial join algorithms [11] like PBSM [21] are mostly appropriate when the geometries are relatively small compared to the tiles, as in the coarse-grained grid of Silk-spatial. In contrast, in the fine-grained grid of GIA.nt, each geometry typically participates in many, small tiles (e.g., see Figure 1). In this way, GIA.nt's Filtering allows for extracting the co-occurrence patterns that are crucial for the progressive methods and the weighting schemes that lie at their core. These patterns, though, cannot be extracted from the Filtering step of most spatial join techniques, including PBSM. Moreover, GIA.nt is more memory efficient than PBSM and other spatial join techniques, as it does not require loading both input datasets in main memory. For these reasons, we exclusively compare GIA.nt with RADON, which achieved the highest time efficiency in Reference [26].

[14] http://flink.apache.org/.

All these approaches operate in a batch (i.e., budget-agnostic) way that consumes the input datasets in no particular order. Inevitably, 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 and HopsWorks.

Instead, we focus on pay-as-you-algorithms that try to optimize the processing order of geometry pairs within a limited budget of temporal or 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 Reference [14] to avoid accessing and buffering detailed representations as much as possible. A progressive version of PBSM for streaming data is proposed in Reference [31], using statistics to determine which contents to keep in the memory buffer and which in disk during evaluation, to maximize the join throughput.

It is worth stressing at this point that a considerable portion of geospatial data on the LOD Cloud involves noise, such as digitization errors and sliver polygons [12, 23]. This noise, which might be part of the original data or might be ingested during triplification or knowledge extraction, inevitably forces Geospatial Interlinking to populate the LOD cloud with incorrect topological relations. However, the correctness of the input geometries is orthogonal to the functionality of the batch and progressive algorithms proposed in this work. Error correction techniques like those discussed in Reference [23] can be used as a pre-processing, data cleaning step that reduces noise either manually or by leveraging ontologies. Our approaches can be applied after preprocessing, directly to the refined data.

Finally, a problem related to this work is *Progressive Entity Resolution* [20, 29, 33], where a pay-as-you-go approach is used to detect matches, i.e., entity profiles describing the same real-world object. For example, *Progressive Sorted Neighborhood* [33] 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, A, 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 Reference [29]: 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 current sliding window. Both approaches are *static*, defining a fixed processing order, independently of the detected matches. A *dynamic* approach adjusts the processing order on-the-fly [20]: if position $A(i, j)$ corresponds to a match, then the positions $A(i + 1, j)$ and $A(i, j + 1)$ are examined next.

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 [26]:

(1) *LineStrings* 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).
(2) *Polygons* constitute, in the simplest case, 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)** [3, 4, 7] defines 10 main topological relations (see Figure 1 for examples):

(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 t lies inside s 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.

From the above definitions, it follows that if `Contains`(s,t) then `Covers`(s,t), too. Similarly, if `Within`(s,t) then `Covered_by`(s,t). Also, it follows that the set of relations `Covers`, `Covered_by`, `Equals`, `Touches`, `Crosses`, `Overlap`, and `Disjoint` are *jointly exclusive and pairwise disjoint*. Finally, whenever a relation other than `Disjoint` holds between a pair of geometries, the relation `Intersects` holds between them as well.

In the following, we disregard the `Disjoint` relation, because it scales quadratically, as the vast majority of pairs typically pertains to unrelated geometries (see Table 4). This means that trillions of `Disjoint` links need to be generated, when the input comprises few million geometries. In contrast, all other topological relations scale linearly with the size of the input data (see Table 4). `Disjoint` is also uninteresting, because it provides no practical linking between entities. It can be inferred, though, from `Intersects` [27]: geometries that are not associated with the latter relation are disjoint.

Overall, we restrict Geospatial Interlinking to identifying the topological relations 1 to 9, which we call **non-trivial**.

3.1 Problem Definition

Every topological relation r is a predicate evaluating to true or false. Thus, Geospatial Interlinking is defined as [26, 27]:

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. These can be derived with simple logical conditions from the **Intersection Matrix** [26], 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 the **dimension** of the intersection $\cap$ of the interior I, boundary B, and exterior E of the geometries s and t. For empty intersections, dim is -1 or F (False), while for non-empty ones, dim is equal to 0 in the case of a point, 1 for a line segment and 2 for 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.[15] For example, `Within` is defined as $IM(0,0) = T \wedge IM(0,2) = F \wedge IM(1,2) = F$ or equivalently as: $\begin{bmatrix} T*F \\ **F \\ *** \end{bmatrix}$. We should avoid, though, to compute redundant relations. For example, if `Contains`(s,t) holds, then `Within`(t,s) is always true [26] and only the former should be added to the LOD cloud. The same applies to `Covers`(s,t) and `CoveredBy`(t,s).

On this basis, we can minimize the cost of Geospatial Interlinking 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 typically addressed in two steps [27, 30]:

(1) Filtering applies space tiling to reduce the computational cost to geometries with MBRs co-occurring in one or more tiles.
(2) Verification computes the Intersection Matrix of the candidate matches 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 [2].

3.2 Progressive Geospatial Interlinking

In this work, we also examine methods that solve the task of Holistic Geospatial Interlinking 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 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 [33] (cf. Figure 2):

- *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.
- *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, then the former should detect significantly more **qualifying geometry pairs**, i.e., geometry pairs that satisfy at least one topological relation $r \in R$.

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

$$PGR(R) = \sum_{i=1}^{|P|} p_Q^i / \left| P_Q^{BU} \right|,$$

[15] See https://en.wikipedia.org/wiki/DE-9IM#Spatial_predicates for more details.

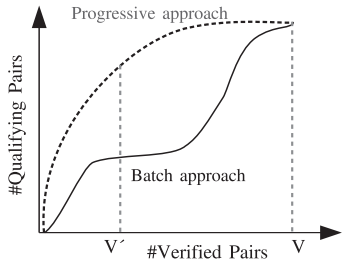

Fig. 2. Evolution of the number of topologically related pairs (vertical axis) as more pairs are verified (horizontal axis) using a progressive and a batch approach.

where $P \subseteq S \times T$ is the set of **candidate pairs**, i.e., the distinct geometry pairs that pass the Filtering step, $|P|$ is its size (i.e., the total number of candidate pairs), $P_Q^{BU} \subseteq P$ is the *largest possible* 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 ith candidate pair. PGR takes values in $[0, 1]$, with higher values indicating higher effectiveness.

In this context, we formally define 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 resources, maximize $PGR(R)$, given the available resources.

To address this task, the progressive methods aim at optimizing the processing order of geometry pairs so that the qualifying ones are verified before the non-related ones. Depending on the way they define the processing order, these methods are distinguished into two categories:

(1) The *Static Progressive Methods* a-priori define an immutable processing order that relies exclusively on the likelihood of every pair to satisfy at least one topological relation. Weighting schemes based on heuristics are used to assess this likelihood (see Section 5.1).
(2) The *Dynamic Progressive Methods* start with a processing order that is determined by the same weighting schemes as the static methods. Yet, they update the initial order on-the-fly, based on the latest outcomes of Verification to continuously promote the pairs that are more likely to satisfy a relation in view of the new links.

4 BATCH ALGORITHM

Before studying progressiveness, we introduce ***Geospatial Interlinking At large*** **(GIA.nt)**, a novel batch algorithm, whose functionality appears in Algorithm 1. Lines 1–12 apply Filtering to index the source dataset, which is the smallest one 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 $\Delta_y = mean_{s \in S} MBR(s).length$, respectively, by adapting RADON's approach so that it considers only the source dataset. 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).

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 $\lfloor 14/\Delta_x \rfloor = 4 \leq i \leq 7 = \lceil 19/\Delta_x \rceil$ and $\lfloor 60/\Delta_y \rfloor = 30 \leq j \leq 31 = \lceil 61/\Delta_y \rceil$.

ALGORITHM 1: GIA.nt

```
   input : the source dataset S, a reader for the target one rd(T) and the set of non-trivial topological
           relations R
   output: the links L = {(s, r, t)|s ∈ S ∧ t ∈ T ∧ r ∈ R ∧ r(s, t)}
 1 I ← {};                                   // Filtering step – 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
13 L ← {};                                  // Verification step – L: The set of detected links
14 while rd(T).hasNext() do
15     t ← rd(T).next();                                    // The current target geometry
16     C_S ← {};                                // The set of candidate source geometries
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             C_S ← C_S ∪ I.getTileContents(i, j);
21             j ← j + 1;
22         end
23         i ← i + 1;
24     end
25     foreach geometry s ∈ C_S 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;
```

GIA.nt's Verification is applied in Lines 13–31. The next target geometry $t \in T$ is read from the disk (Lines 14 and 15) and the tiles that contain its MBR are estimated (Lines 17–24). For each tile, GIA.nt retrieves the contained source geometries 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).

That complexity of Algorithm 1 appears in Table 1, along with that of the main relevant approaches.

GIA.nt's space complexity amounts to $O(|S| + |L|)$. This is at least 50% lower than RADON and RADON2, i.e., $O(|S| + |T| + |L| + |V|)$, where V stands for the set of verified pairs. The reason is

Table 1. Space and Time Complexity of the Main Approaches for Batch Geospatial Interlinking

	Filtering	Time Complexity Verification	Space Complexity
GIA.nt	$O(\|S\|)$	$O(\|(s,t) : MBR(S) \cap MBR(T) \neq \{\}\|)$	$O(\|S\| + \|L\|)$
RADON [27]	$O(\|S\| + \|T\|)$	$O(\|R\| \cdot \|(s,t) : MBR(S) \cap MBR(T) \neq \{\}\|)$	$O(\|S\| + \|T\| + \|L\| + \|V\|)$ or $O(\|S\| \cdot \|T\|))$
RADON2 [1]	$O(\|S\| + \|T\|)$	$O(\|(s,t) : MBR(S) \cap MBR(T) \neq \{\}\|)$	$O(\|S\| + \|T\| + \|L\| + \|V\|)$ or $O(\|S\| \cdot \|T\|))$
stLD [24, 25]	$O(\|I\|)$	$O(\|I\| \cdot \|T\| + \|R\| \cdot \|(s,t) : MBR(S) \cap MBR(T) \neq \{\}\|)$	$O(\|S\| + \|L\| + \|I\|)$ or $O(\|S\| \cdot \|T\|))$

that GIA.nt indexes and maintains in main memory only the source dataset, which is the smallest by definition (i.e., $|S| \leq |T|$). In contrast, RADON and RADON2 store both input datasets in memory. Given that GIA.nt reads (and verifies) the target geometries one by one from the disk, its tiles, which are kept in memory, contain only source geometries—no target ones. GIA.nt also avoids all redundant pairs inherently, due to its geometry-centric Verification: for every target geometry, it aggregates the distinct source geometries in the tiles intersecting its MBR. In contrast, the tile-centric Verification of RADON/RADON2 avoids the redundant candidate pairs by storing all verified ones in memory, increasing the space complexity by $O(|V|)$.

The difference between GIA.nt and stLD is much smaller, as the latter also stores in main memory and places in the tiles of its index only the source geometries. However, stLD has a higher memory footprint, because it maintains a mask for every tile, i.e., a (possibly complex) polygon that captures the empty space inside every tile. The corresponding space complexity is denoted by $O(|I|)$, where $|I|$ stands for the size of the Equigrid, i.e., the number of its tiles. For fine-grained indices, such as those reported in Table 6, $O(|I|)$ is much higher than $O(|S| + |L|)$. Note also the memory requirements of stLD might be even higher, depending on the method that discards redundant candidate pairs—the tiles of its index have overlapping contents, as every geometry is placed into multiple tiles.

It should be stressed at this point that unlike GIA.nt, the other three algorithms compute the relation `disjoint`, too. In this case, their space complexity rises to $O(|S| \cdot |T|)$, as the vast majority of source geometries are typically unrelated to any target geometry. For instance, this applies to all datasets in Table 4. GIA.nt ignores `disjoint`, based on the closed-world assumption: any pair of geometries that are not linked with any positive topological relation are disjoint.

The time complexity of Algorithm 1 for Filtering amounts to $O(|S|)$, since it iterates once over the source geometries. Compared to RADON and RADON2, it is at least 50% faster, because both go through all input geometries, i.e., $O(|S| + |T|)$, to index them and to compute heuristics for switching inputs. stLD employs a static Equigrid, which is defined manually, on the basis of domain knowledge, without processing the input geometries. However, after indexing the source geometries, stLD creates the mask of every tile. The corresponding computational cost is $O(|I|)$, which might be higher than $O(|S|)$ for fine-grained indices.

Regarding Verification, GIA.nt's time complexity is the same as RADON2, which is dominated by the number of candidate pairs, i.e., geometries with intersecting MBRs: $O(|(s,t) : MBR(S) \cap MBR(T) \neq \{\}|)$. For RADON and stLD, the Verification cost amounts to $O(|R| \cdot |(s,t) : MBR(S) \cap MBR(T)|)$, where R denotes the set of topological relations. The reason is that they repeat the same process for every topological relation in R, instead of simultaneously computing all of them. stLD also involves the cost of computing the `overlap` relation between the mask of every tile b and the overlap of every target geometry with b. In the worst case, this cost amounts to $O(|I| \cdot |T|)$, which might be higher than $O(|(s,t) : MBR(S) \cap MBR(T) \neq \{\}|)$ for fine-grained indices.

Finally, it is worth stressing that unlike RADON and RADON2, GIA.nt is easily adapted to massive parallelization according to the MapReduce paradigm, as explained in Section 6. The reason is that 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 PROGRESSIVE ALGORITHMS

The progressive methods extend the batch ones with one more step, operating as follows:

(1) Filtering is first applied to reduce the computational cost to geometries with MBR(s) located in the same tile(s).
(2) *Scheduling* then defines the processing order of the candidate pairs produced by Filtering to maximize Progressive Geometry Recall. It does not verify any pair, as it exclusively relies on heuristics.
(3) Verification eventually examines the candidate pairs. For static methods, this order is specified by Scheduling, while for dynamic ones, this order depends both on Scheduling and on the outcomes of the pairs verified so far.

Note that the second step conveys a delay in the production of results, whose duration we call *scheduling time*, t_s.

Below, we first examine the weighting schemes that lie at the core of Scheduling and then present our static and dynamic progressive approaches.

5.1 Weighting Schemes

To schedule the verification of geometry pairs in a way that maximizes Progressive Geometry Recall, the progressive methods assign a weight to every pair that is analogous to its likelihood to satisfy a non-trivial topological relation. To this end, they employ functions, called *weighting schemes*, which receive as input a pair of source and target geometries, s and t, and produce a score as output. The higher the score, the higher is the likelihood that s and t are a qualifying pair and the earlier they should be processed.

In this context, we consider two types of weighting schemes:

(1) The *hit probability schemes* consider exclusively the tiles shared by s and t. They assume that the more tiles s and t have in common, the more likely they are to be topologically related and, thus, the higher should be their weight.
(2) The *complexity schemes* consider the main characteristics of s and t that affect their verification time, namely, their area and their size. The lower the complexity of these characteristics, the higher the resulting score.

Below, we examine the weighting schemes of every category in detail.

Hit Probability Schemes. The following schemes belong to this category:

(1) Inspired from Term-Frequency in Information Retrieval, ***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 intersecting the MBR of geometry k.
(2) ***Jaccard Similarity* (JS)** normalizes the overlap similarity defined by CF, i.e., $JS(s,t) = \frac{|B_s \cap B_t|}{|B_s|+|B_t|-|B_s \cap B_t|}$. In this way, it captures 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 [28], 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 intersecting s in B is the same as the distribution if we exclude the tiles that intersect t. In more detail, it uses the *contingency table* (see Table 2), where $n_{1,1}$ stands for the number of tiles shared by the two geometries, $n_{1,2}$ for the number of tiles intersecting s but not t, $n_{2,1}$ for the number of tiles intersecting t but not s, and $n_{2,2}$ for the number of tiles intersecting neither geometry. These are the observed values, whereas the expected value for each cell of the contingency

Table 2. Contingency Table for Geometries s and t

	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 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: $w_{i,j} = \sum_{i \in \{1,2\}} \sum_{j \in \{1,2\}} \frac{n_{i,j} - m_{i,j}}{m_{i,j}}$.

Complexity Schemes. The following schemes belong to this category:

(1) ***Minimum Bounding Rectangle Overlap*** **(MBRO)** estimates the portion of the area that is shared by the MBRs of s and t, assuming that the larger this portion is, the more likely are s and t to correspond to a qualifying pair. More formally: $MBRO(s,t) = \frac{MBR(s \cap t)}{MBR(s \cap t)} = \frac{MBR(s \cap t)}{MBR(s)+MBR(t)-MBR(s \cap t)}$, where $MBR(g)$ stands for g's MBR.

(2) ***Inverse Sum of Points*** **(ISP)** promotes pairs with simple geometries, which are thus faster to verify. It actually assumes that the fewer the points in the boundary of a geometry, the simpler it is. More formally: $ISP(s,t) = \frac{1}{p(s)+p(t)}$, where $p(g)$ denotes the function that returns the points in the boundary of geometry g.

Note that for MBRO, we considered four additional definitions, namely, $MBR(s \cap t)$, $MBR(s \cap t)/min(MBR(s), MBR(t))$, $MBR(s \cap t)/max(MBR(s), MBR(t))$ and $MBR(s \cap t)/(MBR(s)+MBR(t))$. Preliminary experiments, though, revealed that these variants underperform the selected definition.

Discussion. CF, MBRO, and ISP rely 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 intersect the MBR of a particular target geometry. In contrast, JS and χ^2 require the number of common tiles as well as the total number of *valid* tiles per geometry—a tile is valid if it intersects at least one geometry from each input dataset; that is, a tile containing only source or target geometries should be ignored during the computation of JS and χ^2 weights. This **global information** can be computed only by indexing both datasets, which increases the run-time and complicates the weight estimation in the context of the MapReduce framework (to compute the valid tiles in the MapReduce context, it would require an extra MapReduce job that discovers and aggregates all the valid tiles and then broadcasts this information to all nodes). To overcome this drawback, we skip the ideal case of computing JS and χ^2 accurately for each candidate pair of geometries. Instead, we consider their *approximations*, which replace the actual number of valid 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 nine tiles, but only six of them are valid, intersecting the MBR of a source geometry; the tiles/blocks b_{03}, b_{13}, and b_{23} intersect 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.

A crucial aspect for the performance of weighting schemes is the **distinctiveness** of their scores. The more distinctive they are, the lower is the portion of ties, thus yielding a more deterministic approach. In other words, a weighting scheme that assigns the same probability

to different unlabeled comparisons yields a random ordering that depends on the order of appearance of geometry pairs or other non-controllable parameters.

Most weighting schemes suffer from poor distinctiveness (see Section 7.2.4). To address it, we consider two approaches:

(1) The *composite schemes* combine two of the aforementioned, atomic schemes in the following way: the *primary* one is used for scheduling all pairs, while the *secondary* one is used for resolving the ties of the main one.
(2) The *hybrid schemes* combine a hit probability scheme with a complexity one through division or multiplication in an effort to leverage both types of evidence.

We assess the relative performance of both approaches in Section 7.2.6.

5.2 Static Progressive GIA.nt

This algorithm turns GIA.nt into a progressive approach by gathering in a min-max priority queue the *BU* most promising pairs of input geometries. To populate this queue, it 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, then (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, Static Progressive GIA.nt produces a global ordering of pairs.

The details of its functionality are outlined in Algorithm 2. Lines 1–12 apply Filtering, which indexes the source dataset in the same way as GIA.nt. Next, Scheduling creates three data structures:

(1) the min-max priority queue T_C (Line 13) stores the top-*BU* weighted pairs in decreasing order of weight. Its head, retrieved by *pop*(), always contains the pair with the minimum weight, while its tail, retrieved by *popLast*(), contains the pair with maximum weight.
(2) the `int` array $flags$ (Line 14) designates the id of the target geometry that was last associated with a specific source geometry.
(3) the `int` array $frequency$ (Line 14) counts the frequency 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.

These two arrays facilitate the computations of the hit probability schemes. For example, $frequency[1] = 5$ and $flags[1] = 2$ mean that the second source geometry (id = 1) shares five tiles with the third target geometry (id = 2).

The three data structures are populated by the loop in Lines 15–45. Note that the arrays $flags$ and $frequency$ are only required if a hit probability weighting scheme is selected. In case of MBRO and ISP, they can be omitted, along with the corresponding operations in Lines 23, 24, and 27. Lines 19–21 identify the tiles that should intersect the MBR of 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), then the arrays $flags$ and $frequency$ are updated accordingly (Lines 23 and 24) and s_n is added in the set of candidate related geometries C_S (Line 25). Next, the co-occurrence frequency with t_m is incremented (Line 27). The distinct candidate geometries are then weighted (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, then its

ALGORITHM 2: Static 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 and the weighting scheme W
output: the links $L = \{(s, r, t) | s \in S \wedge t \in T \wedge r \in R \wedge r(s, t)\}$

1 $I \leftarrow \{\}$; // **Filtering step** – I: Equigrid index structure
2 $(\Delta_x, \Delta_y) \leftarrow$ defineIndexGranularity(S);
3 ...; /* Index S as in Algorithm 1, Lines 1-12 */
13 $minWeight$ = 0.0 ; $T_C \leftarrow \{\}$; // **Scheduling step** – T_C: Priority queue
14 $flags[] \leftarrow \{\}$; $frequency[] \leftarrow \{\}$;
15 **while** $rd(T)$.*hasNext()* **do**
16 $t_m \leftarrow rd(T)$.next(); // The current target geometry
17 $C_S \leftarrow \{\}$; // The set of candidate source geometries
18 $(x_1(t_m), y_1(t_m), x_2(t_m), y_2(t_m)) \leftarrow$ getDiagC(t_m);
19 **for** $i \leftarrow \lfloor x_1(t_m)/\Delta_x \rfloor$ **to** $\lceil x_2(t_m)/\Delta_x \rceil$ **do**
20 **for** $j \leftarrow \lfloor y_1(t_m)/\Delta_y \rfloor$ **to** $\lceil y_2(t_m)/\Delta_y \rceil$ **do**
21 **foreach** $s_n \in I$.*getTileContents(i, j)* **do**
22 **if** $flags[n] \neq m$ **then**
23 $flags[n] = m$;
24 $frequency[n] = 0$;
25 $C_S \leftarrow C_S \cup s_n$;
26 **end**
27 $frequency[n]$++;
28 **end**
29 $j \leftarrow j + 1$;
30 **end**
31 $i \leftarrow i + 1$;
32 **end**
33 **foreach** *geometry* $s_n \in C_S$ **do**
34 **if** *intersectingMBRs(s, t)* **then**
35 $w_{s,t} \leftarrow$ weight($frequency[n]$, s_n, t_m);
36 **if** $minWeight < w_{s,t}$ **then**
37 T_C.add($\{s, t\}$, $w_{s,t}$);
38 **if** $BU < T_C$.*size()* **then**
39 $head = T_C$.pop();
40 $minWeight = head$.getWeight();
41 **end**
42 **end**
43 **end**
44 **end**
45 **end**
46 $L \leftarrow \{\}$; // **Verification step** – L: The set of detected links
47 **while** $T_C \neq \{\}$ **do**
48 $tail = T_C$.popLast();
49 $IM \leftarrow$ verify($tail.s$, $tail.t$);
50 $L \leftarrow L \cup IM$.getRelations();
51 **end**
52 **return** L;

head is removed and the minimum weight threshold is updated accordingly (Lines 38–41). Finally, Verification examines the pairs in the priority queue in decreasing order of weight, iteratively retrieving its tail (Lines 47 and 48). 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 and 50).

The complexity of Algorithm 2 is discussed in Section 5.3.

It is worth noting at this point that another static progressive method, called Progressive RADON, is presented in Reference [19]. As its name suggests, it adapts RADON to a pay-as-you-go functionality through a local, tile-based operation. Its Scheduling orders the tiles in decreasing or increasing size and Verification sorts the candidate pairs inside every tile in decreasing weight. We omit the details of this approach, due to its poor performance [19].

5.3 Dynamic Progressive GIA.nt

The core idea of this approach is that whenever a new pair of geometries (s, t) is detected as qualifying, we boost the weight of all candidate pairs that are associated with s and t and are still located in the priority queue so that they are verified earlier. This is useful for example in cases where the source dataset involves long LineString geometries like roads, whereas the target dataset involves Polygon geometries buildings: the more buildings a road touched so far, the higher should be the weight of the rest of the candidate buildings, as it is likely a main road.

In this context, Dynamic Progressive GIA.nt implements the same approach as its static counterpart for the Filtering and Scheduling steps. It only modifies the Verification step, as shown in Algorithm 3. More specifically, before verifying any pair of geometries, it populates two hash maps with the candidate pairs for every source and target geometry in the priority queue (Lines 46–54). The keys of the first hash map (H_S) correspond to the ids of the source geometries, with every value for key k encompassing all geometry pairs in the priority queue with k as their source geometry id (Lines 48–50). The second hash map (H_T) does the same for the target geometries (Lines 51–53). As a result, these two data structures associate every geometry with all its pairs in the priority queue, allowing for quickly updating their weights, whenever a new qualifying pair is detected.

This is efficiently implemented by the subsequent loop (Lines 56–79). First, the next top-weighted pair of geometries, i.e., the tail of the priority queue, is verified in Lines 57–59. If this pair yields at least one topological relation (Line 60), then the set of detected links is updated accordingly (Line 61). Next, the weights of candidate pairs c_s associated with the verified source geometry ($tail.s$) are updated in Lines 62–69. Every candidate pair p is removed from the priority queue (Line 64), with an operation that returns `true` if p was contained in the queue and `false` otherwise. In the latter case, no action is required, as p has already been verified. In the former case, the weight of p is incremented as follows (Line 66): $w = w_o \cdot (1 + q)$, where w_o stands for the original weight, as determined by the selected weighted scheme, and q for the number of times its source and its target geometries, $p.s$ and $p.t$, respectively, have been involved in an already detected qualifying pair. In each turn, the maximum value for q is 1, as either $p.s$ or $p.t$ participate in the latest qualifying pair p' (if both geometries participated in p', then $p \equiv p'$, meaning that the priority queue contains duplicates, which is not the case). The updated pair is then added again to the priority queue (Line 67). The same processing is then applied to the candidate pairs of the target geometry, $tail.t$ (Lines 70–77).

It is worth noting at this point that we need to minimize the cost of the operations on the priority queue to ensure a low overhead on the overall verification time, t_v. To this end, a *tree set* is used for the implementation of the priority queue, as the time complexity of both *add* and *remove* is $O(\log n)$. In contrast, Static Progressive GIA.nt employs a min-max priority queue, whose time complexity for the operation *remove* is linear, $O(n)$.

ALGORITHM 3: Dynamic 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 and the weighting scheme W
output: the links $L = \{(s, r, t) | s \in S \wedge t \in T \wedge r \in R \wedge r(s, t)\}$

```
1  ...;                          /* Filtering and Scheduling steps as in Alg. 2, Lines 1-45 */
46 H_S ← {}; H_T ← {};  // Verification step – H_S, H_T: Hash tables with the candidate pairs per geometry
47 foreach geometry pair p ∈ T_C do
48     H_S ← H_S.get(p.s);                          // Update candidates of source geometry
49     H_S ← H_S ∪ p;
50     H_S.put(p.s, H_S);
51     H_T ← H_T.get(p.t);                          // Update candidates of target geometry
52     H_T ← H_T ∪ p;
53     H_T.put(p.t, H_T);
54 end
55 L ← {};                                          // The set of detected links
56 while T_C ≠ {} do
57     tail = T_C.popLast();
58     IM ← verify(tail.s, tail.t);
59     l ← IM.getRelations();
60     if l ≠ {} then
61         L ← L ∪ l;
62         H_s ← H_S.get(tail.s);                   // Get candidates of source geometry
63         foreach geometry pair p ∈ H_s do
64             exists ← T_C.remove(p);
65             if exists then
66                 p.incrementWeight();
67                 T_C.add(p);
68             end
69         end
70         H_t ← H_T.get(tail.t);                   // Get candidates of target geometry
71         foreach geometry pair p ∈ H_t do
72             exists ← T_C.remove(p);
73             if exists then
74                 p.incrementWeight();
75                 T_C.add(p);
76             end
77         end
78     end
79 end
80 return L;
```

The complexity of Algorithm 3 is qualitatively compared with that of Algorithm 2 in Table 3. For both algorithms, the time complexity for Filtering amounts to $O(|S|)$, as they simply iterate once over all source geometries. The time complexity of their Scheduling amounts to $O(|C| \cdot \log |BU|)$, where $|C|$ denotes the total number of candidate pairs with intersecting MBRs and $\log |BU|$ is the worst-case cost of inserting every one of these pairs in the priority queue. During Verification,

Table 3. Space and Time Complexity of Static and Dynamic Progressive GIA.nt

	Time Complexity			Space Complexity
	Filtering	Scheduling	Verification	
Static Progressive GIA.nt (Algorithm 2)	$O(\|S\|)$	$O(\|C\| \cdot \log \|BU\|)$	$O(\|BU\|)$	$O(\|S\| + \|L\| + \|BU\|)$
Dynamic Progressive GIA.nt (Algorithm 3)	$O(\|S\|)$	$O(\|C\| \cdot \log \|BU\|)$	$O(\|BU\| \cdot (1 + \bar{f_s} + \bar{f_t})\|)$	$O(\|S\| + \|L\| + \|BU\|)$

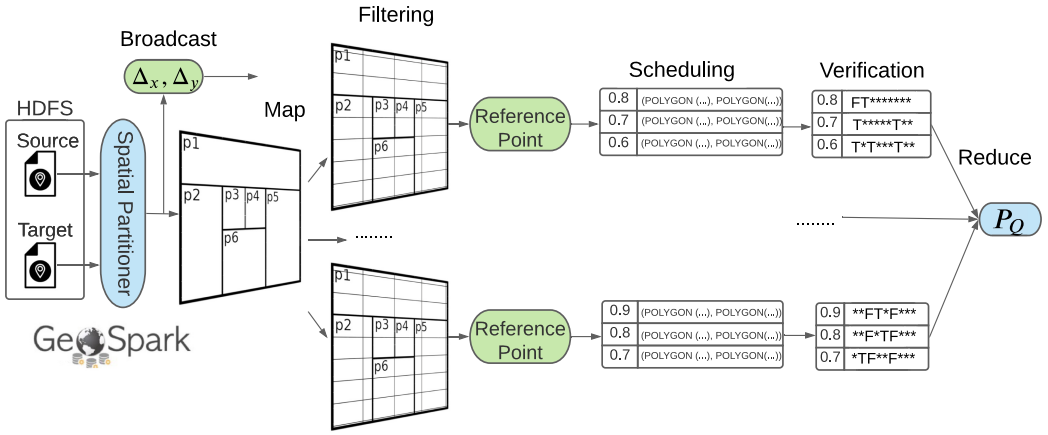

Fig. 3. Parallel (Progressive) GIA.nt on top of Apache Spark. Both datasets are loaded in HDFS and spatially partitioned using GeoSpark's Quad-Tree. Map assigns each partition to an Executor, which applies Filtering to find the candidate pairs and the reference point technique to discard the redundant ones. For Progressive GIA.nt, Scheduling orders locally (i.e., partition-wise) 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.

Algorithm 2 merely goes through the selected top-weighted pairs, having a time complexity of $O(|BU|)$. Algorithm 3 adds to this computational cost the overhead of weight updating, which amounts to $O(|BU|(\bar{f_s} + \bar{f_t}))$, where $\bar{f_s}$ ($\bar{f_t}$) stands for the average frequency of a source (target) geometry in the top-BU weighted candidate pairs. That is, $\bar{f_s}$ and $\bar{f_t}$ indicate the average number of candidate pairs that are updated in Lines 63–69 and 71–77 of Algorithm 3, respectively. Regarding the space complexity, it amounts to $O(|S|+|L|+|BU|)$ for Algorithm 2, which maintains in memory the smallest input dataset along with the set of detected links and the top-BU weighted candidate pairs. In theory, Algorithm 3 has the same time complexity, but in practice, its memory footprint is slightly higher, due to the hash tables H_S and H_T, whose sizes are equal to the budget, i.e., $O(|BU|)$. Given, though, that they merely store geometry ids (rather than the geometries themselves), their impact on memory requirements is kept to the minimum.

6 MASSIVE PARALLELIZATION

We now explain how to parallelize (Progressive) GIA.nt according to the MapReduce framework.

Our approach is outlined in Figure 3. Initially, it loads both datasets as RDDs that are spatially partitioned across the available workers, based on GeoSpark's Quad-Tree [35]. This Quad-Tree is built based on a sample of the source geometries. The source and target RDDs are partitioned using the same partitioner, such that the topologically close geometries belong to partitions with the same partition id. In case a geometry crosses the border between two partitions, it is replicated and placed into both partitions. 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.

Before joining the RDDs of source and target geometries in each partition, the granularity of space tiling is estimated. We use the same tile dimensions as in serial GIA.nt. This requires the computation of $\Delta_x = mean_{s \in S} MBR(s).width$ and $\Delta_y = mean_{s \in S} MBR(s).length$, which is carried out through a single MapReduce job: each Executor sums the extends of its local source geometries during the Map phase, while the Reduce phase aggregates the results and computes the tile dimensions. The Driver then broadcasts Δ_x and Δ_y to the Executors. This process triggers the evaluation of the execution plan of the source RDD (i.e., loading, partitioning, and other transformations); to avoid re-calculating it for each subsequent action, we cache it locally, in the main memory of each Executor.[16]

In this context, GIA.nt starts by sending to each Executor a partition of the input data 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* candidate source geometries in these tiles and verifies their topological relations with t. To avoid duplicate verifications, given that every geometry intersects multiple tiles, GIA.nt employs the reference point technique [6] to ensure that each candidate pair is verified only in the partition and tile that contains the top left corner of the corresponding MBR intersection. All qualifying pairs are aggregated during the Reduce phase.

For Static Progressive GIA.nt, during the Map phase, every Executor receives as input a partition of both input datasets and applies Filtering to index the source geometries. Then, it applies Scheduling, processing the target geometries one by one to estimate their weights with the intersecting source geometries. Each partition stores its top-k weighted pairs in a min-max priority queue, 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. After all target geometries have been processed, the Verification phase merely computes the intersection matrix for the top-k weighted candidate pairs in the local priority queue. The qualifying pairs of each Executor are aggregated by the Reduce phase.

The implementation of Dynamic Progressive GIA.nt is similar. During the Map phase, it applies the same Filtering and Scheduling steps to discover the locally top-k weighted pairs in each partition. Then, it populates the local hash tables, H_S and H_T, based on the discovered top-k candidate pairs. These hash tables allow for quickly updating all the candidate pairs that are associated with the latest qualifying pair. Subsequently, the Verification step is initiated. Whenever a new qualifying pair is detected, H_S and H_T are queried for the related candidate pairs that have not be verified so far. These pairs are removed from the local priority queue and reinserted with a boosted weight. Finally, the qualifying pairs detected by 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 to promote concurrency, making the most of massive parallelization.

[16]In Apache Spark, an execution plan is composed of a series of transformations (i.e., mapping, filtering, partitioning, etc.) and an action (i.e., count, sum, reduce, etc.). The transformations are lazy in the sense that they are not executed until an action invokes them and their results are dumped, by default. Users can maintain the results of a specific transformation in the memory or on the disk by caching it.

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 distinguish the partitions into *overloaded* and *well-balanced* ones. The former are defined as those with a number of source geometries significantly higher than the average one across all partitions. To detect them, we use the Z-score,[17] which measures how many standard deviations a value is away from the mean value. In this context, a partition is considered to be overloaded if its Z-score exceeds a predefined threshold, set to 2.5. The rest of the partitions are marked as well-balanced and are processed as described above. After completing their processing, 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, without missing any candidate pairs. Redundant pairs are again discarded with the reference point technique. Overall, this is an effective and efficient load balancing strategy as long as it applies to a small portion of the input data, and not to the entire dataset, given that it requires the replication of each entity as many times as the numbers its tiles.

7 EXPERIMENTAL ANALYSIS

We now present the experiments that assess the effectiveness, time efficiency, and memory footprint of our approaches. Section 7.1 presents the setup of our experimental study, while Section 7.2 focuses on the experiments of the serial processing. In more detail, we compare our batch algorithm, GIA.nt, with RADON/RADON2 in Section 7.2.1, verifying the superiority of the former in all respects. In Section 7.2.2, we compare Static Progressive GIA.nt with Optimal and Random Scheduling, demonstrating that *MBRO* excels in effectiveness and *ISP* in time efficiency. We further delve into the performance of Static Progressive GIA.nt in Sections 7.2.3 and 7.2.4; the former investigates the recall of each weighting scheme per topological relation, while the latter examines every weighting scheme with respect to its distinctiveness, the diversity of its top-weighted geometries as well as their area and number of boundary points. Section 7.2.5 demonstrates the superiority of Dynamic Progressive GIA.nt over its static counterpart, while Section 7.2.6 verifies the superiority of composite weighting schemes, which use *MBRO* as a secondary scheme. The massive parallelization of our approaches is examined in Section 7.3: We show that Parallel (batch) GIA.nt significantly outperforms GeoSpark in Section 7.3.1, we report the performance of all parallel approaches over the largest dataset pair in Section 7.3.2, and we conclude with their scalability analysis in Section 7.3.3. Finally, we discuss the main findings in Section 7.4.

7.1 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 @ 2.2 GHz, 128 GB RAM, running Ubuntu 14.04.5 LTS. For RADON's implementation, we used LIMES version 1.7.1.[18] For all time measurements, we used a single physical core and performed three repetitions, reporting the average. For the verification of geometry pairs, we used the JTS Topology Suite, version 1.16.[19]

All parallel methods and experiments were implemented in Scala 2.12 using Spark 2.4.4. Most of the experiments were performed on a Hadoop cluster consisting of a single node with 32 cores

[17] https://en.wikipedia.org/wiki/Standard_score.
[18] https://github.com/dice-group/LIMES.
[19] https://github.com/locationtech/jts.

Table 4. Technical Characteristics of the Real Pairs of Datasets for Geospatial Interlinking

	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	8,326,942	9,831,432	19,592,688	72,339,926
#Target Geometries	5,838,339	19,592,688	9,831,432	72,339,926	70,380,191	114,796,567
Cartesian Product	$1.34 \cdot 10^{13}$	$4.49 \cdot 10^{13}$	$8.19 \cdot 10^{13}$	$7.11 \cdot 10^{14}$	$1.38 \cdot 10^{15}$	$8.30 \cdot 10^{15}$
Candidate Pairs	6,310,640	15,729,319	19,595,036	67,336,808	430,597,631	257,075,645
#Qualifying Pairs	2,401,396	199,122	3,841,922	12,145,630	163,982,138	1,041,562
`#Contains`	806,158	3,792	267,457	5,147,704	53,758,453	274,953
`#CoveredBy`	0	0	1,944,207	47,253	12,218,868	82,828
`#Covers`	832,843	4,692	267,713	5,284,672	53,758,453	274,966
`#Crosses`	40,489	106,823	217,198	5,700,257	6,769	313,566
`#Equals`	0	0	61,712	2,047	12,218,868	18,909
`#Intersects`	2,401,396	199,122	3,841,922	12,145,630	163,982,138	1,037,153
`#Overlaps`	0	0	488,814	42,331	73	54,810
`#Touches`	1,554,749	88,507	986,522	1,210,230	110,216,843	331,166
`#Within`	0	0	1,943,643	47,155	12,218,868	81,567
Total Topological Relations	5,635,635	402,936	10,019,188	29,627,279	418,379,333	2,481,027

Intel Xeon CPU E5-4603 v2 @ 2.20 GHz[20] and 128 GB DDR3 RAM, 1.6 Tb mechanical disk. The experiment presented in Figure 12 (left) was performed on a small cluster that runs the Hopsworks data platform [10]. The main module of Hopsworks is Hops,[21] which is a next generation distribution of Apache Hadoop, using a novel implementation of HDFS, called HopsFS [18]. This cluster consists of a two workers with 32 Intel Xeon CPU E5-2650 v3 at 2.30 GHz and 86 GB DDR3 RAM each. Unless stated otherwise, the experiments performed on the single node used 16 Executors with two cores each and 7 GB of memory, and the experiments performed on the two nodes, used 15 Executors with four cores each and 10 GB of memory.

Datasets. The technical characteristics of the real datasets we use in our experiments are reported in Table 4. All of them have been widely used in the literature [8, 32] and are publicly available.[22] They contain public data about area hydrography (AREAWATER), linear hydrography (LINEARWATER), roads (ROADS), and all edges (EDGES) in USA. They 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 4 shows statistics for a pair (D_1–D_6) of interlinked datasets. Note that for D_3 and D_4, the number of relations is lower than those reported in Reference [19], as we now exclude the entities that correspond to geometry collections, because their verification is not supported by the JTS library. Note also that in D_1, D_2, and D_4, the source geometries are Polygons and the target ones LineStrings, and vice versa for D_6. In contrast, D_3 and D_5 are homogeneous, as they exclusively pertain to Polygons and LineStrings, respectively.

In the following, Section 7.2 elaborates on the serial experiments that were carried out over the five smallest pairs of datasets, D_1 to D_5. The largest dataset pair, D_6, does not fit into the main memory of our stand-alone server. It is used only in the parallel experiments that are presented in Section 7.3.

7.2 Serial Processing

7.2.1 Batch Geospatial Interlinking. Table 5 compares the performance of GIA.nt and RADON with respect to filtering time (t_f), verification time (t_v), and memory footprint (m). Note that

[20] The system uses hyper-threading hence it has 16 physical cores.
[21] https://github.com/hopshadoop/hops.
[22] http://spatialhadoop.cs.umn.edu/datasets.html.

Table 5. The Filtering Time (t_f), the Verification Time (t_v), and the Memory Footprint (m) of RADON and GIA.nt

	D_1			D_2			D_3			D_4			D_5		
	t_f (s)	t_v (min)	m (GB)	t_f (s)	t_v (min)	m (GB)	t_f (s)	t_v (h)	m (GB)	t_f (s)	t_v (h)	m (GB)	t_f (s)	t_v (h)	m (GB)
RADON	56	78.1	64	240	174.5	78	—	—	—	—	—	—	—	—	
GIA.nt	42	76.7	22	42	166.4	22	257	9.5	80	53	29.3	68	36	18.1	85

For GIA.nt, t_v includes the time required for reading the target dataset from the disk, which amounts to 4.9, 6.4, 5.9, 18.3, and 20.1 min for D_1, D_2, D_3, D_4, and D_5, respectively.

Table 6. Technical Characteristics of the Equigrid Indices Constructed by RADON's and GIA.nt's Filtering

	RADON		GIA.nt	
	D_1	D_2	D_1	D_2
#Dimensionality ($X \times Y$)	68,493 × 20,999	105,656 × 32,580	138,325 × 43,559	138,325 × 43,559
#Total Tiles	$1.44 \cdot 10^9$	$3.44 \cdot 10^9$	$6.03 \cdot 10^9$	$6.03 \cdot 10^9$
#Populated Tiles	$4.72 \cdot 10^7$	$1.51 \cdot 10^8$	$5.34 \cdot 10^7$	$4.76 \cdot 10^7$
#Pairs in Tiles	$9.69 \cdot 10^7$	$2.40 \cdot 10^8$	$1.49 \cdot 10^8$	$3.05 \cdot 10^8$
#Unique Pairs	$2.34 \cdot 10^7$	$3.35 \cdot 10^7$	$1.23 \cdot 10^7$	$2.80 \cdot 10^7$

RADON can only process D_1 and D_2 with the available memory resources (128 GB), while GIA.nt is able to process all datasets except the largest one, D_6.

We observe that for both algorithms, Verification is the bottleneck, with Filtering accounting for a negligible portion of the overall run-time. Yet, Filtering reduces the tens of *trillions* pairs considered by the brute-force approach to tens of *millions* candidate pairs, as shown in Table 6. This renders it 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. Table 6 indicates that GIA.nt yields a space tiling of higher granularity 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 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 and use the same Verification algorithm, which is implemented by the JTS library. Note that RADON2 is not implemented in LIMES, but we approximate its holistic geospatial interlinking through the topological relation `intersects`. We observe that GIA.nt is slightly faster than RADON, 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. Two are the reasons for the high time efficiency of GIA.nt: (i) it handles every geometry through its id—rather than its URI, as in RADON, and (ii) it uses 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.[23]

These implementation optimizations also reduce GIA.nt's memory footprint to a significant extent. In combination with the fact that the target dataset is not loaded in main memory, GIA.nt consistently occupies at least 66% less main memory than RADON in Table 5.

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

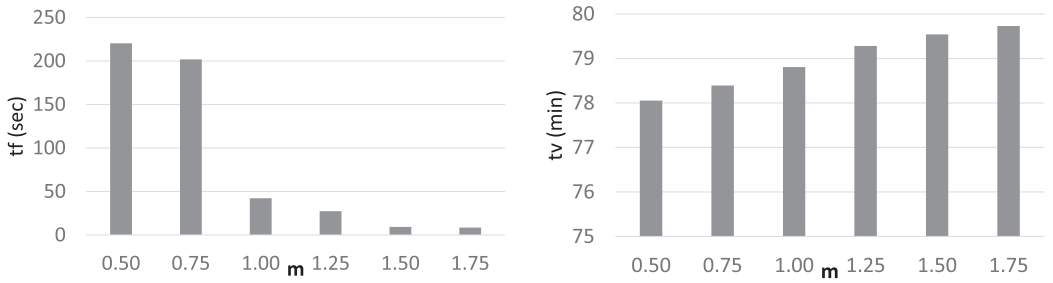

Fig. 4. The evolution of GIA.nt's filtering and verification time (t_f on the left and and t_v on the right, respectively) over D_1 as we multiply the average width and length of its tiles with m.

Finally, Table 5 suggests that as the input size increases from 8 to 90 million geometries, GIA.nt's run-time increases linearly, from ∼1.5 to ∼18 h, respectively. Of course, the run-time fluctuates significantly among the dataset pairs, depending on the complexity of their geometries: the more complex they are, the more time-consuming is their Verification. We also observe that GIA.nt's memory footprint scales sublinearly, from 22 GB for D_1 to 85 GB 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.

Overall, *we can conclude that GIA.nt improves RADON in all respects. For a slightly lower running time, the memory footprint is 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.*

Index Granularity. It is interesting to examine at this point the impact of index granularity on the time efficiency of GIA.nt. To this end, we multiplied the average length and width, which determine the dimensions of the Equigrid tiles, with a parameter $m \in (0, 2)$. We actually considered all values of m in $[0.50, 1.75]$ with a step of 0.25. The corresponding filtering and verification times of GIA.nt over D_1 appear in Figure 4.

We observe that the filtering time decreases as we increase m. The reason is that the smaller m gets, the smaller are the resulting tiles and, most importantly, the higher is their number. In fact, $m = 0.50$ and $m = 0.75$ yield an extremely large, fine-grained index, which places every source geometry into too many tiles (note that for $m = 0.25$, the index got so large that did not fit within the available memory). The opposite is true for the remaining values of m, especially for $m = 1.5$ and $m = 1.75$, which yield a rather coarse-grained index.

However, the impact of m on the verification time is negligible: as m increases from 0.5 to 1.75, t_v increases by just 2.1%, from 78.1 to 79.7 min. Three are the reasons for this: (i) t_v includes the time required for reading the target geometries from the disk, which is independent of the index granularity, (ii) t_v is dominated by the time required to compute the Intersection Matrix between all candidate pairs with intersecting MBRs. Their number is also independent of the index granularity. (iii) The index granularity affects only the number of candidate pairs whose MBRs are checked for intersection. The more coarse-grained the index is (i.e., the larger m is), the higher is their number and the more time is spent on checking MBR intersection. This explains the 1.6 min increase in t_v between $m = 0.5$ and $m = 1.75$.

Overall, we can conclude that the index granularity plays a minor role in the overall run-time of GIA.nt, since it solely affects t_f, which accounts for less than 1% of the overall run-time (only for $m = 0.5$ and $m = 0.75$, this raises up to ∼4.5%). The verification time, the bottleneck of Geospatial Interlinking, is affected to an insignificant extent, mainly because the number of verified pairs is determined by the

Table 7. Performance of Static Progressive GIA.nt for All Weighting Schemes in Comparison to the Optimal (Opt.) and the Random (Rnd.) Approach for Budgets of 5M and 10M Verifications Across All Datasets

		BU = 5M							BU = 10M						
		Opt.	Rnd.	Progressive GIA.nt					Opt.	Rnd.	Progressive GIA.nt				
				CF	*JS*	χ^2	*MBRO*	*ISP*			*CF*	*JS*	χ^2	*MBRO*	*ISP*
D_1	PGR	0.760	0.396	0.299	**0.655**	0.648	0.498	0.623	0.880	0.500	0.415	**0.722**	0.716	0.565	0.694
	Recall	1.000	0.792	0.726	0.949	0.946	0.751	0.921	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Precis.	0.480	0.380	0.349	0.456	0.454	0.361	0.442	0.381	0.381	0.381	0.381	0.381	0.381	0.381
	t_s (min)	—	—	5.9	5.5	6.2	4.9	4.7	—	—	4.0	4.7	4.5	4.6	4.4
	t_v (h)	—	—	1.1	0.3	0.3	0.5	0.1	—	—	1.3	1.3	1.3	1.3	1.3
D_2	PGR	0.980	0.159	0.481	0.541	0.507	**0.598**	0.234	0.990	0.318	0.609	0.715	0.689	**0.733**	0.371
	Recall	1.000	0.318	0.647	0.804	0.777	0.803	0.369	1.000	0.636	0.822	0.946	0.936	0.913	0.679
	Precis.	0.040	0.013	0.026	0.032	0.031	0.032	0.015	0.020	0.013	0.016	0.019	0.019	0.018	0.014
	t_s (min)	—	—	5.8	5.9	6.8	6.7	6.3	—	—	5.9	5.9	6.2	6.7	6.0
	t_v (h)	—	—	1.1	0.3	0.3	0.5	0.1	—	—	2.0	1.0	1.0	1.4	0.5
D_3	PGR	0.616	0.128	0.122	0.264	0.264	**0.274**	0.237	0.808	0.255	0.242	0.460	**0.461**	0.449	0.417
	Recall	1.000	0.255	0.239	0.453	0.454	0.467	0.428	1.000	0.510	0.496	0.830	0.833	0.761	0.761
	Precis.	0.768	0.196	0.184	0.348	0.349	0.359	0.329	0.384	0.196	0.191	0.319	0.320	0.292	0.292
	t_s (min)	—	—	9.3	8.8	9.4	9.0	9.5	—	—	9.1	9.6	9.4	10.4	9.5
	t_v (h)	—	—	3.2	0.2	0.2	0.4	0.1	—	—	5.4	0.8	0.9	1.7	0.6
D_4	PGR	0.500	0.037	**0.192**	0.079	0.070	0.153	0.054	0.500	0.074	**0.172**	0.078	0.074	0.136	0.055
	Recall	1.000	0.074	0.361	0.165	0.151	0.285	0.109	1.000	0.148	0.309	0.151	0.152	0.246	0.113
	Precis.	1.000	0.074	0.361	0.165	0.151	0.285	0.109	1.000	0.148	0.309	0.151	0.152	0.246	0.113
	t_s (min)	—	—	25.4	26.3	27.4	26.4	27.0	—	—	26.3	26.8	28.0	31.9	28.7
	t_v (h)	—	—	1.2	0.04	0.4	0.1	0.03	—	—	2.9	0.1	0.4	0.2	0.1
D_5	PGR	0.500	0.058	0.121	0.462	0.462	**0.500**	0.485	0.500	0.116	0.124	0.462	0.461	**0.500**	0.483
	Recall	1.000	0.116	0.247	0.923	0.923	1.000	0.971	1.000	0.232	0.260	0.925	0.925	1.000	0.956
	Precis.	1.000	0.116	0.247	0.923	0.923	1.000	0.971	1.000	0.232	0.260	0.925	0.925	1.000	0.956
	t_s (min)	—	—	26.3	23.9	26.7	27.2	25.6	—	—	26.4	26.1	27.0	27.9	24.8
	t_v (min)	—	—	31.2	2.4	2.4	3.4	0.8	—	—	55.2	4.9	4.9	7.2	1.8

intersecting MBRs. We also observe that the default configuration of GIA.nt's index (m = 1.0) yields a good trade-off between t_f and t_v, minimizing the overall run-time.

7.2.2 Static Progressive Geospatial Interlinking. To evaluate the effectiveness of progressive methods, we consider the following three measures:

(1) Progressive Geometry Recall (*PGR*—see Section 3.1),
(2) Recall = $P_Q^{D,BU}/P_Q^{BU}$, and
(3) Precision = $P_Q^{D,BU}/BU$,

where $P_Q^{D,BU}$ and P_Q^{BU} stand for the number of detected qualifying pairs and the maximum possible number of qualifying pairs *in the given budget BU* (i.e., maximum number of permitted verifications), respectively. All measures are defined in $[0, 1]$, with higher values indicating higher effectiveness.

To evaluate the time efficiency of progressive methods, 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 Table 5.

As baseline methods, we consider **Optimal Scheduling**, which verifies all qualifying pairs before the non-qualifying candidate ones, and **Random Scheduling**, which corresponds to a batch approach that does not specify a deterministic processing order for the input data (it lacks the Scheduling step). Instead, its output rate is random, depending on the order of appearance of the candidate pairs. For this reason, we assess its effectiveness through the average value for *PGR*, Recall and Precision over 100 random permutations of the candidate pairs.

Table 7 reports the performance of all methods over D_1-D_5 for two different budgets: 5 and 10 million verifications, i.e., *BU* = 5M and *BU* = 10M, respectively. Note 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 7(b) achieve perfect recall and precision over D_1, while there are significant variations in *PGR*, due to the different processing order that is defined by each method.

Regarding effectiveness, we observe that MBRO achieves the most effective scheduling in most datasets, particularly for D_2, D_3, and D_5, for both budgets. Its *PGR* is 18.4% (12.3%) higher than the second best one (JS) for BU = 5M (BU = 10M), on average, across all datasets, even when considering D_1, where MBRO underperforms JS by <20%, due to the prevalence of the relation `touches`. Excluding D_1, MBRO also achieves the highest average Precision and Recall for both budgets. JS lies consistently at the second place, with its average Recall and Precision being lower by 9.0% (2.4%) and 14.2% (10.2%) than that of MBRO for BU = 5M (BU = 10M), respectively.

There are several exceptions, though:

(1) In D_1, JS is the top performing weighting scheme, while MBRO's performance is consistently low. The reason is that the MBRs of ~500,000 qualifying pairs share very small areas, yielding a weight lower than 10^{-5}. This is expected given that the relation `Touches` is the prevalent one in this dataset.
(2) In D_3, χ^2 is the top performing scheme for $BU = 10M$, with MBRO following in close distance. This is because MBRO identifies more pairs than χ^2 for the relations `covered-by` and `within` when $BU = 5M$, but this is reversed for $BU = 10M$. Given that D_3 is the only dataset pair that exclusively interlinks Polygons, we can conclude that the co-occurrence patterns captured by χ^2 are more crucial than the overlap of MBRs for capturing all qualifying pairs of this type.
(3) In D_4, CF is the top performing weighting scheme, with MBRO being the second best scheme. This should be attributed to the very large number of `Crosses` relations in this dataset pair, unlike all others. Apparently, the number of common tiles plays the most decisive role in identifying this topological relation.

Comparing Progressive GIA.nt with the two baselines, we observe that MBRO is the only weighting scheme that matches the effectiveness of Optimal Scheduling, yet only in D_5. In all other cases, there is significant room for improving the performance of Progressive GIA.nt. *MBRO is also the only weighting scheme that consistently outperforms Random Scheduling.* CF underperforms Random Scheduling in D_1 and D_3, regardless of the budget, while the rest of the weighting schemes do so in D_4 for BU = 10M.

The relative effectiveness of Static Progressive GIA.nt and the two baseline methods is visualized in the diagrams of Figure 5, which show the evolution of qualifying pairs with respect to the candidate ones over all datasets for BU = 10M. We observe that the area under the curve of Random Scheduling is consistently lower than that of Static Progressive GIA.nt, except for very few cases, such as CF in D_1. We also observe that *the lower the portion of qualifying pairs to the candidate ones in a dataset pair (see Table 4), the larger is the difference between the area under the curve of Optimal scheduling and the best configurations for Static Progressive GIA.nt.*

Regarding time efficiency, we observe that *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 5 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. In fact, the effect of ties can be observed in t_s for D_1, where the second budget is larger than the number of candidate pairs. As a result, there are no ties for BU = 10M, thus yielding a lower t_s than BU = 5M. In all other datasets, t_s remains

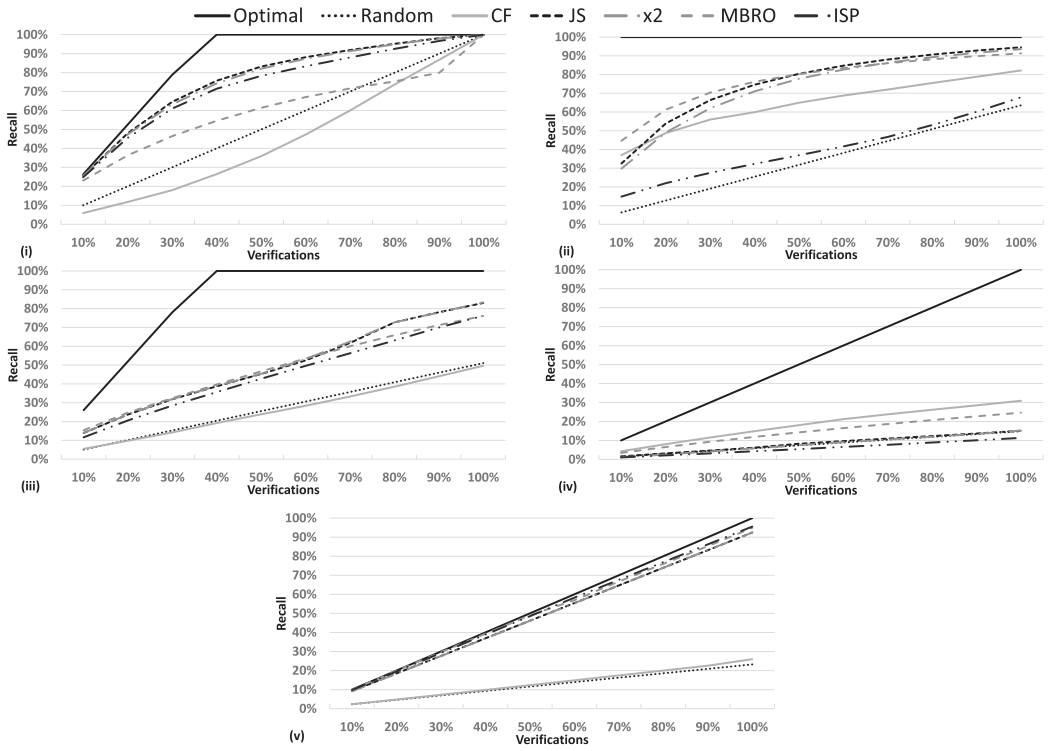

Fig. 5. The evolution of Recall (on the vertical axis) with respect to the number of verifications (on the horizontal axis) for the Optimal approach, the random one as well as for Static Progressive GIA.nt in combination with all weighting schemes. In all cases, BU = 10M. All dataset pairs are considered: (i) D_1, (ii) D_2, (iii) D_3, (iv) D_4, and (v) D_5.

relatively independent of the budget, as the larger number/cost of insertions counterbalances the lower number/cost of ties.

However, the bottleneck in Progressive GIA.nt is the Verification step. We observe that *ISP consistently achieves the lowest t_v across all datasets, regardless of the budget.* This is expected as it selects the simplest geometry pairs. In most cases, though, JS follows in close distance, with χ^2 in the third place.

Most importantly, *Static Progressive GIA.nt offers a significantly better trade-off between effectiveness and time efficiency than RADON2.* The overall run-times of the three top-performing weighting schemes for BU = 5M are significantly lower than that of RADON2 over D_1: JS and χ^2 are faster by 53%, while *ISP* is faster by 67%. Yet, they all achieve a recall well above 90%. A similar pattern is observed for BU = 10M over D_2: MBRO requires half of RADON2's run-time to detect 91.3% of all topological relations, while JS and χ^2 detect >93.5% of all relations, while being faster by 61%.

7.2.3 Recall Per Topological Relation. In this section, we examine the recall of the five weighting schemes in combination with Static Progressive GIA.nt with respect to each topological relation for both budgets we have considered (5M and 10M pairs). In this way, we can detect whether a particular scheme is ideal for a specific relation.

We can define *recall per topological relation* by modifying the definition of recall in Section 7.2.2 to account only for pairs satisfying the topological relation in question.

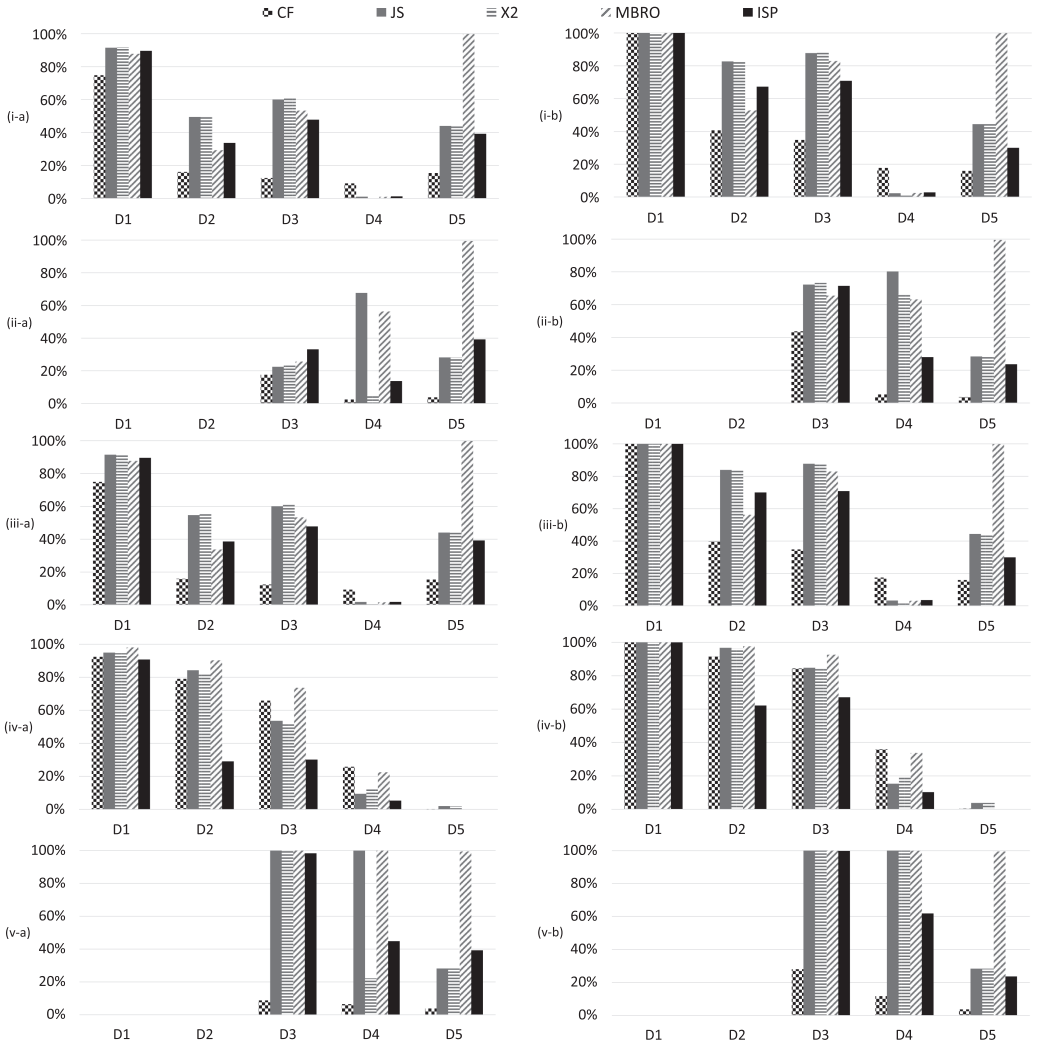

Fig. 6. Recall per weighting scheme and topological relation across all dataset pairs: (i) `contains`, (ii) `covered-by`, (iii) `covers`, (iv) `crosses`, and (v) `equals`. The left column corresponds to BU = 5M and the right one to BU = 10M.

The recall per topological relation of Static Progressive GIA.nt for (a) BU = 5M pairs and (b) BU = 10M pairs in combination with every weighting scheme is presented in Figures 6 and 7, in the left and right columns, respectively. We now examine the rate of detecting a particular positive relation of the DE-9IM model in comparison with the overall recall. The latter, which is presented in Table 7, actually corresponds to the recall of `intersects`, which is the cornerstone relation for every qualifying pair of geometries as we explained in Section 3—any other positive relation is satisfied iff `intersects` is satisfied, too. For this reason, the number of `intersects` relations per dataset in Table 4 is higher than all other topological relations and equals the number of qualifying pairs.

In this context, the relative recall between `intersects` and a topological relation r does not pertain to the absolute number of detected links, but to their rate of detection. For a specific budget

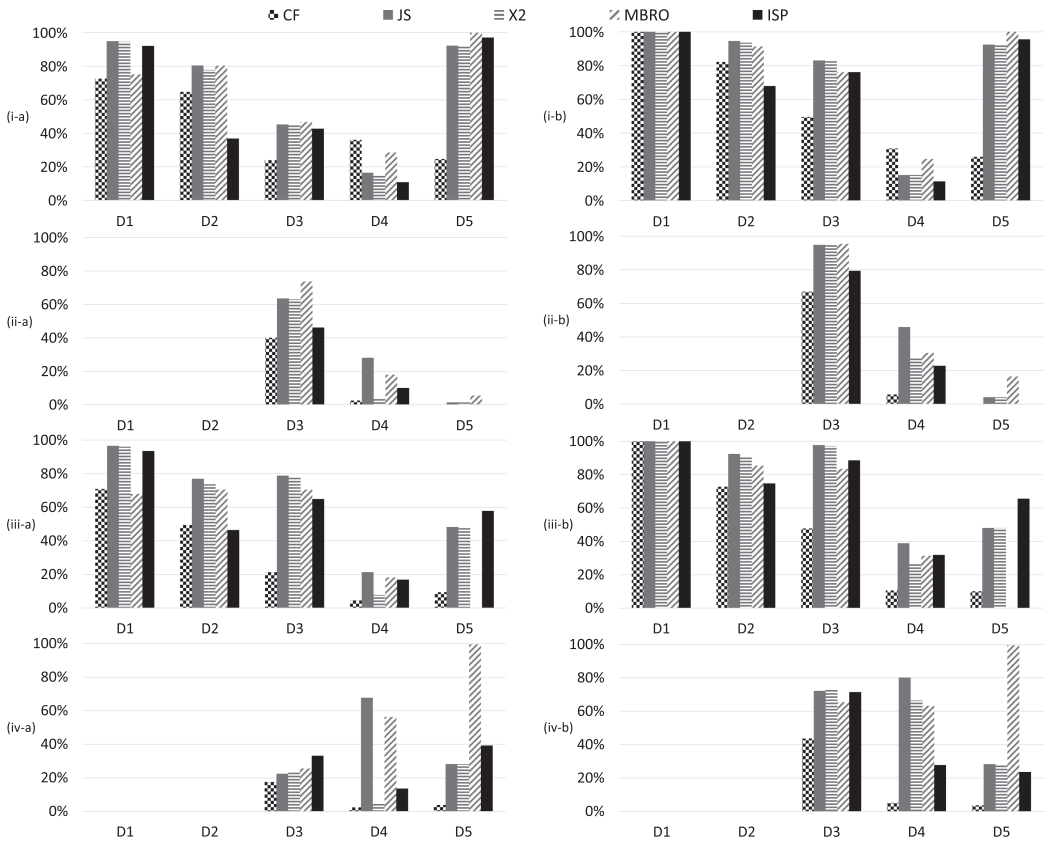

Fig. 7. Recall per weighting scheme and topological relation across all dataset pairs: (i) intersects, (ii) overlaps, (iii) touches, and (iv) within. The left column corresponds to BU = 5M and the right one to BU = 10M.

BU and a specific weighting scheme w, if the recall of relation r is higher than that of intersects, a higher proportion of the existing r links has been detected after BU verifications, because the w is apt to detect the relation r. For example, Table 4 reports that D_3 involves 3.8M intersects and 1.9M covered-by relations; for BU = 5M, if the recall of a weighting scheme is 80% for the former and 90% for the latter, it means that after 5M verifications, it has detected 3.1M intersects and 1.8M covered-by links, respectively. Yet, the rate of detecting covered-by within BU is higher than that of intersects.

Regarding CF, we observe that its recall for all relations is typically much lower than that of intersects. The only exception is the relation crosses, whose relative recall is consistently higher. In the case of $D3$, it is actually 176% (70%) higher for BU = 5M (BU = 10M), as shown in Figure 6(iv). For $D5$, CF's recall for crosses is less than 0.5%, since there are just ~6,800 relations among the ~164 million qualifying pairs (see Table 4). Another exception is the overlap relation in $D3$, whose relative recall is 67% (35%) higher than intersects for BU = 5M (BU = 10M). This does not apply to the other datasets, as the portion of overlap links among the qualifying pairs tends to zero. We can conclude that *CF is ideal for the relations* crosses *and* overlap*, provided that there is a sufficient portion of such links in the input datasets.*

The rest of the weighting schemes exhibit correlated performance that depends on the dataset at hand and the type of geometries it involves. For D_3, which interlinks Polygons (Lakes) with

Polygons (Parks), all weighting schemes except CF, exhibit much higher recall than `intersects` for the relations `overlaps`, `touches` and `equals`. On the contrary, their recall is much lower for the relations `covered-by` and `within`. For D_5, which interlinks LineStrings (ROADS) with LineStrings (EDGES), the recall of all relations is much lower than `intersects` for all weighting schemes except MBRO. This scheme maintains almost perfect recall for all relations except `crosses`, `overlaps` and `touches`. Finally, for D_4, which interlinks Polygons (Parks) with LineStrings (Roads), the four weighting schemes achieve significantly higher recall than `intersects` for the relations `covered-by`, `equals` and `within`. Contrariwise, their recall is significantly lower for the relations `contains` and `covers`. The latter patterns are verified in D_2, too, which also interlinks Polygons (AREAWATER) with LineStrings (ROADS). All other cases do not allow for drawing safe conclusions, as there are inconsistent patterns among the two different budgets or there is complete lack of some relations (in D_1 and D_2) or the largest budget is larger than the number of qualifying pairs (in D_1).

Overall, we can conclude that the effectiveness of each weighting scheme in detecting a particular topological relation depends largely on the type of geometries (LineStrings or Polygons) in the source and target datasets.

7.2.4 Weighting Scheme Characteristics. To explain the relative performance of the five weighting schemes in combination with Static Progressive GIA.nt, we investigate the main features that account for their effectiveness and efficiency.

In Figure 8(i), we observe the **distinctiveness** of the scores produced by each weighting scheme across all dataset pairs for both budgets—BU = 5M on the left and BU = 10M on the right. Higher values indicate more distinctive weights, thus yielding a lower portion of ties, with 100% suggesting that a different score is assigned to every pair. Therefore, higher distinctiveness corresponds to a more deterministic approach. This behavior corresponds only to MBRO, which consistently exceeds 80%. It is followed by χ^2 in a large distance, which fluctuates between 1% and 10%. All other weighting schemes exhibit low distinctiveness, yielding a random ordering of the top-weighted pairs that might depend on the order of appearance or the data structure lying at the core of the approach. The situation is worse for CF and ISP, whose distinctiveness is consistently close to zero. For JS, distinctiveness is zero only over D_5, where every selected pair is assigned the same (maximum) score. This applies to all other weighting schemes, though, for this particular dataset. *These patterns call for composite weighting schemes with MBRO as the secondary approach.*

Figures 8(ii) and 8(iii) report the **source** and **target diversity**, respectively, per weighting scheme and dataset, with BU = 5M corresponding to the left column and BU = 10M to the right one. In other words, these diagrams measure the portion of different source and target geometries in the selected top-weighted pair for each budget. The higher this portion is, the larger is the scope of a weighting scheme in the sense that its search space includes a larger part of the input data. This is especially helpful in datasets with a large number of qualifying pairs, such as D_5. For this dataset, MBRO achieves the maximum diversity, much higher than those of the other weighting schemes, for both source and target geometries across both budgets. This accounts for its perfect effectiveness (PGR = 0.5), which is also much higher than the rest of the weighting schemes (see Table 7). In most other cases, the source and target diversity of all weighting schemes are quite similar, with CF constituting the only consistent exception. Its source diversity is consistently the lowest by far across all datasets (except for D_1 and BU = 10M, where all schemes verify all candidate pairs). Its target diversity is also rather low in most cases. This probably accounts for its consistently low effectiveness, as its search space is restricted to the largest geometries, which are not necessarily involved in qualifying pairs. The only exception is D_4, where CF achieves the top performance among all weighting schemes.

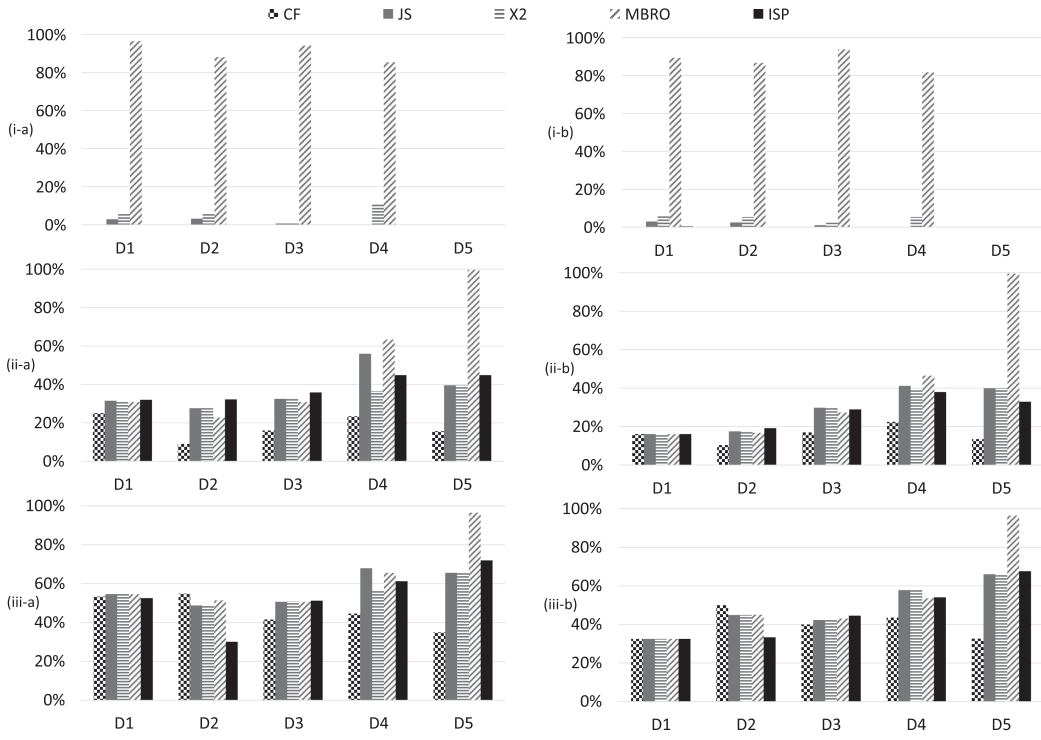

Fig. 8. (i) The distinctiveness of the scores produced by each weighting scheme and the diversity of the (ii) source and (iii) target geometries selected by each scheme. The left column corresponds to BU = 5M and the right one to BU = 10M.

The outlying behavior for CF is verified by the diagrams in Figure 9, which analyze the complexity statistics of the top-weighted geometry pairs. In Figure 9(i), we observe the average area of the selected source geometries takes very high values for CF for both budgets. For D_3, the average area is actually much higher than the maximum value of the vertical axis: 14.7 for BU = 5M on the left and 13.0 for BU = 10M on the right. The second largest areas, with a large difference though, correspond to MBRO, with the remaining weighting schemes selecting geometries of rather small areas. Similar patterns appear in Figure 9(ii), which presents the average area of target geometries. The only difference is that ISP selects the second largest geometries in D_2 and D_3 for BU = 5M and the largest ones for the same datasets for BU = 10M. These target geometries correspond to parks, whose borders are defined by very few boundary points, but their area is quite large. This is verified in Figures 9(iii) and 9(iv), which report the average number of boundary points per source and target geometry for all weighting schemes, datasets and budgets. As expected, ISP yields the geometries with the simplest borders for all datasets, with the only exception corresponding to D_2, where the selected target geometries are the (second) most complex ones for BU = 10M (BU = 5M). This is counterbalanced, though, by the very low number of points in the borders of the source geometries for the same dataset. Among the other weighting schemes, there is clear pattern: CF consistently selects the most complex ones across all datasets and budgets, with MBRO following in the second place, while JS and χ^2 lie between MBRO and ISP. The only exception is D_2, as mentioned above. These complexity characteristics account for the relative Verification times of the five weighting schemes in Table 7.

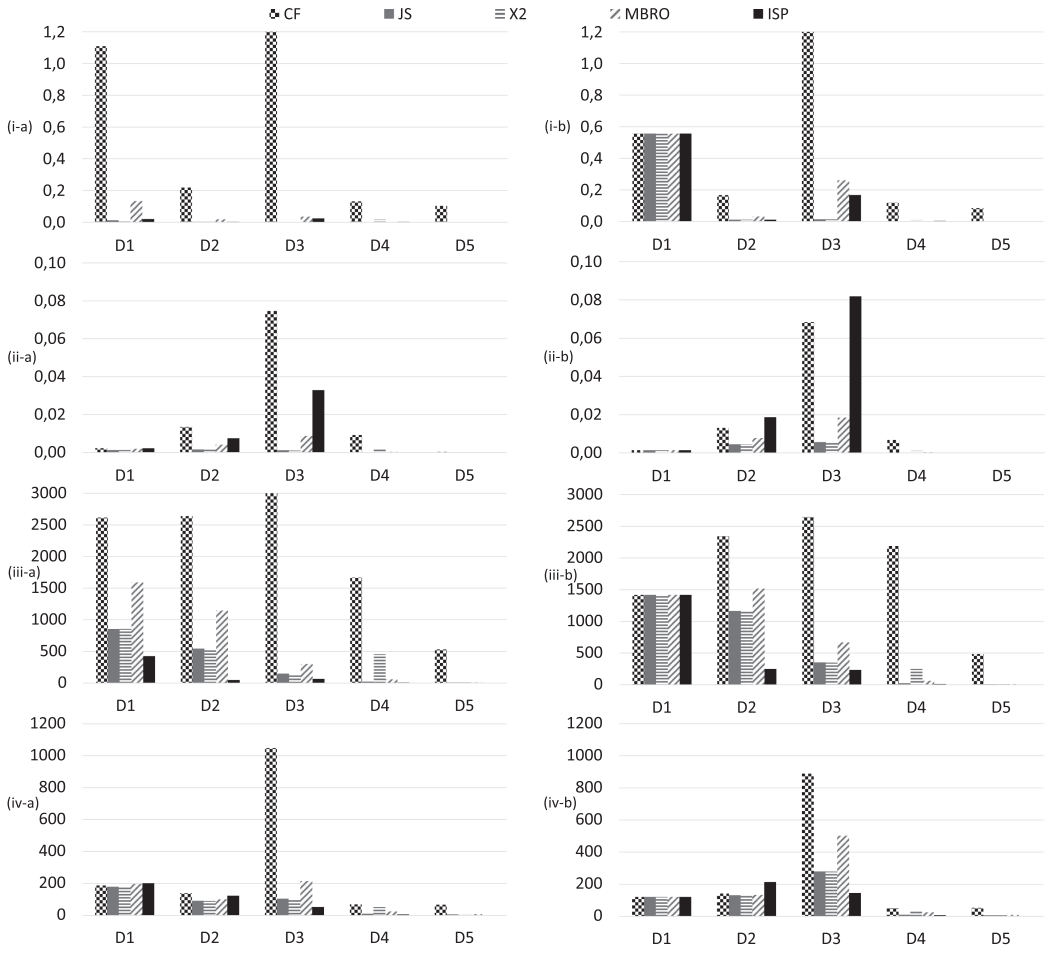

Fig. 9. Complexity statistics of the geometry pairs that are selected by each weighting scheme: the average area per (i) source and (ii) target geometry as well as the average number of boundary points per (iii) source and (iv) target geometry. The left column corresponds to BU = 5M and the right one to BU = 10M.

7.2.5 Dynamic Progressive Geospatial Interlinking. By default, the recall and precision of this algorithm remain the same as that of Static Progressive GIA.nt, as they both operate on the same top-BU weighted candidate pairs. The effectiveness of the two algorithms differs only with respect to *PGR*. The change in *PGR* is reported in Figure 10(i), with the left diagram corresponding to BU = 5M and the right one to BU = 10M. We observe that *in most cases, there is a significant increase that depends on the dataset at hand*, rising up to 12%, 11%, and 20% for D_2, D_3, and D_4, respectively. For D_1, though, the increase is negligible, up to 2.5% for JS and χ^2. This should be attributed to the high proportion of qualifying pairs over the candidate ones, which yields an already high *PGR* for Static Progressive GIA.nt. The same applies to D_5, where *PGR* is very close to the optimal one for most weighting schemes (see Table 7).

A major role is also played by the geometry pairs selected by each weighting scheme. We observe that for CF, *PGR* is consistently reduced among all datasets. On average, it drops by 9.4% for BU = 5M and by 8.2% for BU = 10M. This should be attributed to its low diversity of source and target

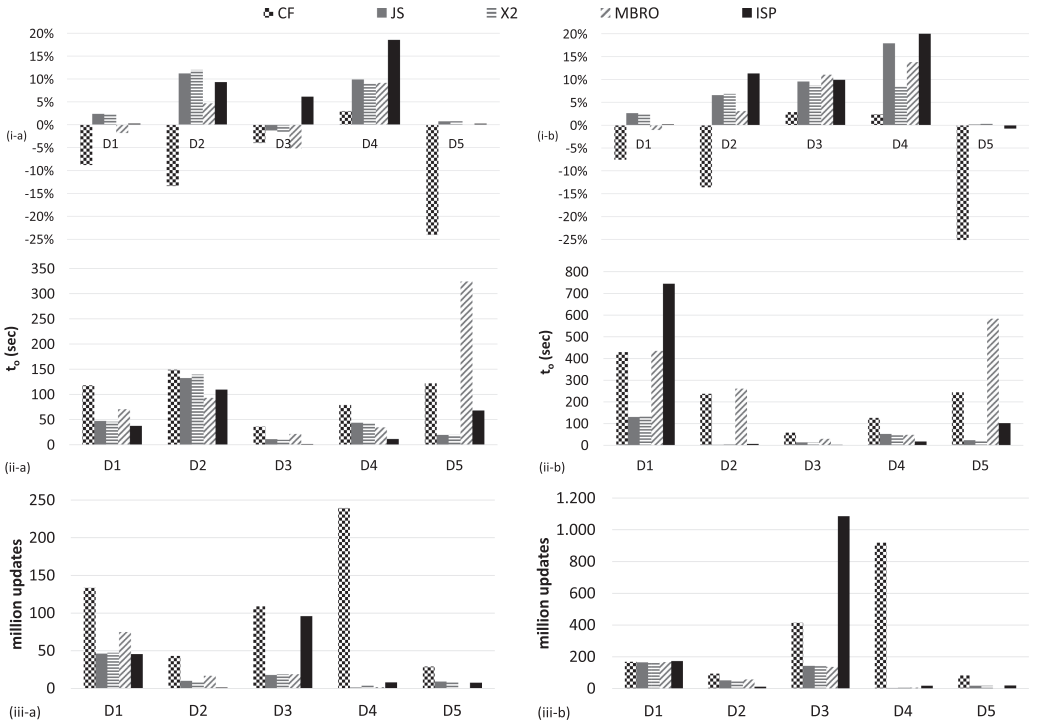

Fig. 10. Performance of Dynamic Progressive GIA.nt with respect to: (i) its impact on the PGR of its static counterpart, (ii) its overhead time to Verification, and (iii) the number of weight update operations. The left column is for BU = 5M and the right one for BU = 10M.

geometries across all datasets and budgets, as indicated in Figures 8(ii) and 8(iii). *The lower these diversities are, the larger is the impact of Dynamic Progressive GIA.nt on its PGR*. In contrast, there is a significant increase in *PGR* for JS, χ^2 and ISP, which on average, across all dataset pairs, amounts to 4.6% (7.4%), 4.5% (5.4%), and 6.9% (8.1%), respectively, for BU = 5M (BU = 10M). For MBRO, this increase is lower, 1.4% for BU = 5M and 5.4% for BU = 10M, due to its poor performance over D_1 (for reasons explained above) and its perfect performance (PGR = 0.5) for D_5.

Regarding the time efficiency of Dynamic Progressive GIA.nt, its indexing and scheduling time is identical with that of its static counterpart. There is an increase only in its verification time, due to the **overhead time** (t_o) required for updating the weights of the geometry pairs in the priority queue. This time is reported in Figure 10(ii). We observe that it consistently amounts to few minutes, *thus being negligible when compared to the overall verification time in each case*. We actually observe that the more time consuming the verification time of a weighting scheme is, the higher is the overhead time. Indeed, in most cases, the highest t_o corresponds to CF, which also exhibits the highest v_t. The actual size of t_o depends on the number of weight updates, which is reported in Figure 10(iii), as well as on the source and target diversities, which are reported in Figures 8(ii) and 8(iii), respectively.

7.2.6 Composite and Hybrid Weighting Schemes. We now examine the effect of composite weighting schemes on the performance of Dynamic Progressive GIA.nt. Due to the high distinctiveness of the scores produced by MBRO, we use it as secondary for all other schemes. We also exclude it from this analysis, as there is no point in combining it with another scheme, due to the very low portion of ties.

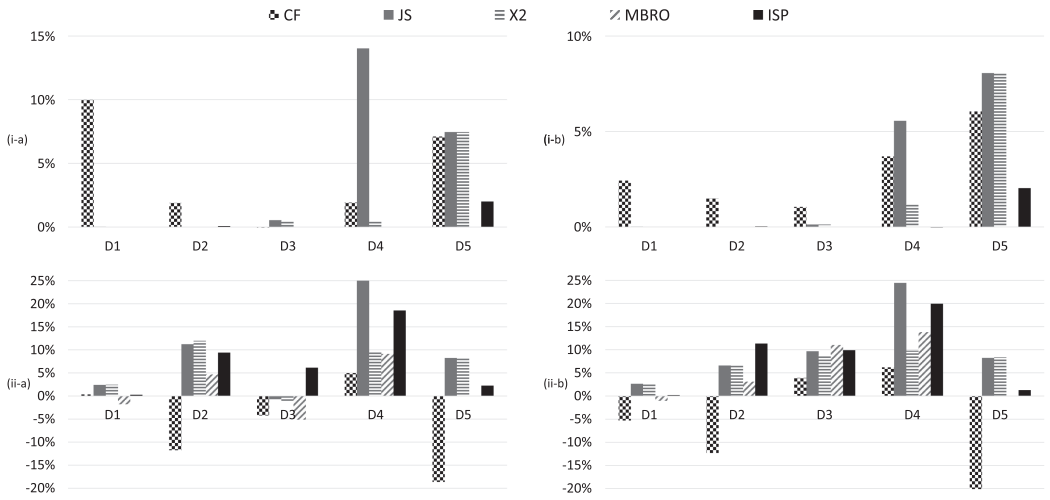

Fig. 11. Performance of composite weighting schemes with respect to: (i) the PGR of Dynamic Progressive GIA.nt and (ii) the PGR of Static Progressive GIA.nt. The left column corresponds to BU = 5M and the right one to BU = 10M.

The impact of composite schemes on Progressive Geometry Recall is reported in Figure 11(i), with BU = 5M corresponding to the left and BU = 10M to the right diagram. (MBRO is absent from both diagrams, as there is no change in its performance.) We observe that there is no case of negative impact—in the worst case, *PGR* remains the same. This applies particularly to datasets like D_2 and D_3, where the composite scheme produces a similar processing order with its primary one. The only exception is CF, which consistently exhibits significant increase, as it suffers from the highest portion of ties. On average, across all datasets, its *PGR* raises by 4.1% for BU = 5M and by 2.9% for BU = 10M. It is followed in close distance by JS, whose *PGR* raises by 4.4% and 2.8% for BU = 5M and BU = 10M, respectively, on average. For χ^2, the average increase is reduced to 1.7% and 1.9%, respectively, while the increase in the case of ISP is negligible, 0.4% on average for both budgets, because most of the pairs it selects are non-qualifying.

It is worth examining at this point the overall increase in *PGR* in comparison with Static Progressive GIA.nt, which is reported in Figure 11(ii). On average, across all datasets for BU = 5M, JS improves by 9.3%, χ^2 by 6.2%, MBRO by 1.4%, and ISP by 7.3%. For BU = 10M, the average improvements are 10.3% for JS, 7.3% for χ^2, 5.4% for MBRO, and 8.5% for ISP. The only exception remains CF, whose *PGR* drops by 5.9% and 5.6%, on average, across all datasets, for BU = 5M and BU = 10M, respectively. Compared to Figure 10(i), i.e., the improvements conveyed by Dynamic Progressive GIA.nt in combination with the original weighting schemes, *PGR* has significantly increased for all weighting schemes, except MBRO, which remains the same, as it cannot be enhanced through a composite scheme, due to its already high distinctiveness.

Given that the composite schemes alter all aspects of Progressive GIA.nt, Table 8 reports the resulting performance with respect to all effectiveness and time efficiency measures. Comparing it with the performance of Static Progressive GIA.nt in Table 7, we observe the following patterns:

(1) For CF, recall and precision increase by 2.6% on average, across all datasets and budgets, even though its *PGR* decreases by 5.8%, on average. In other words, CF's combination with MBRO as a secondary weighting scheme is able to select more qualifying pairs, but CF cannot make the most of dynamic processing, due to its low source and target diversity, which means that it continuously updates the weights of the same candidate pairs.

Table 8. Performance of Dynamic Progressive GIA.nt with Composite Weighting Schemes Based on MBRO in Comparison to the Optimal (Opt.) and the Random (Rnd.) Approach for Budgets of 5M and 10M Verifications Across All Datasets

		BU = 5M							BU = 10M						
		Opt.	Rnd.	Progressive GIA.nt					Opt.	Rnd.	Progressive GIA.nt				
				CF	*JS*	χ^2	*MBRO*	*ISP*			*CF*	*JS*	χ^2	*MBRO*	*ISP*
D_1	PGR	0.760	0.396	0.300	**0.671**	0.664	0.489	0.624	0.880	0.500	0.393	**0.741**	0.734	0.560	0.695
	Recall	1.000	0.792	0.761	0.949	0.946	0.751	0.921	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Precis.	0.480	0.380	0.365	0.456	0.454	0.361	0.442	0.381	0.381	0.381	0.381	0.381	0.381	0.381
	t_s (min)	—	—	5.4	5.2	4.9	4.9	4.6	—	—	5.3	4.9	4.9	5.1	4.9
	t_v (h)	—	—	1.0	0.5	0.5	0.9	0.3	—	—	1.3	1.3	1.3	1.3	1.3
D_2	PGR	0.980	0.159	0.424	0.602	0.568	**0.626**	0.256	0.990	0.318	0.534	**0.762**	0.737	0.755	0.413
	Recall	1.000	0.318	0.657	0.804	0.777	0.803	0.369	1.000	0.636	0.838	0.946	0.936	0.913	0.679
	Precis.	0.040	0.013	0.026	0.032	0.031	0.032	0.015	0.020	0.013	0.017	0.019	0.019	0.018	0.014
	t_s (min)	—	—	6.8	6.7	6.4	7.0	6.7	—	—	6.7	6.6	6.5	7.1	6.6
	t_v (h)	—	—	1.1	0.3	0.3	0.5	0.1	—	—	2.1	1.0	1.0	1.5	0.5
D_3	PGR	0.616	0.128	0.117	**0.262**	0.261	0.260	0.251	0.808	0.255	0.251	**0.505**	0.502	0.499	0.458
	Recall	1.000	0.255	0.240	0.453	0.454	0.454	0.428	1.000	0.510	0.496	0.830	0.833	0.761	0.761
	Precis.	0.768	0.196	0.185	0.348	0.349	0.349	0.329	0.384	0.196	0.191	0.319	0.320	0.292	0.293
	t_s (min)	—	—	10.6	11.0	9.0	9.6	9.2	—	—	11.3	10.9	11.3	10.3	11.4
	t_v (h)	—	—	3.2	0.2	0.2	0.2	0.1	—	—	5.3	0.9	0.9	1.9	0.8
D_4	PGR	0.500	0.037	**0.201**	0.099	0.076	0.167	0.063	0.500	0.074	**0.183**	0.097	0.081	0.155	0.066
	Recall	1.000	0.074	0.364	0.174	0.155	0.285	0.109	1.000	0.148	0.309	0.151	0.152	0.246	0.114
	Precis.	1.000	0.074	0.364	0.174	0.155	0.285	0.109	1.000	0.148	0.309	0.151	0.152	0.246	0.114
	t_s (min)	—	—	28.1	29.2	26.5	25.9	28.1	—	—	29.4	28.5	30.4	31.6	28.8
	t_v (h)	—	—	0.9	0.04	0.4	0.09	0.03	—	—	1.7	0.09	0.4	0.2	0.07
D_5	PGR	0.500	0.058	0.098	**0.500**	**0.500**	**0.500**	0.496	0.500	0.116	0.098	**0.500**	**0.500**	**0.500**	0.489
	Recall	1.000	0.116	0.258	1.000	1.000	1.000	0.965	1.000	0.232	0.290	1.000	1.000	1.000	0.952
	Precis.	1.000	0.116	0.258	1.000	1.000	1.000	0.965	1.000	0.232	0.290	1.000	1.000	1.000	0.952
	t_s (min)	—	—	26.3	26.7	25.3	25.6	24.1	—	—	27.1	27.8	27.9	27.1	27.0
	t_v (min)	—	—	0.5	0.06	0.06	0.07	0.02	—	—	0.9	0.13	0.12	0.13	0.04

(2) All effectiveness measures of JS and χ^2 increase by more than 8% over D_5, since their combination with MBRO as a secondary weighting scheme allows them to select only qualifying pairs, thus matching the performance of the optimal method.
(3) Regarding the scheduling time, there are mixed patterns in the case of *BU* = 5M, but for *BU* = 10M, there is a significant increase for all composite weighting schemes. The more ties there are for the primary scheme, the higher the increase is: on average, across all datasets, t_s raises by 17.2% for CF, by 10.2% for ISP, by 8.5% for χ^2, and by 8.5% for JS. This should be attributed to a higher number of insertions in the priority queue, as ties are now resolved with the help of the secondary scheme.
(4) The verification time of most schemes increases significantly over D_5. On average, for both budgets, it raises by 49.5% for JS, by 47% for χ^2, 14.8% for MBRO, and 22.8% for ISP. The reason is that these datasets involve very simple geometries, whose verification time was already very low, taking few minutes, as shown in Table 7. In this context, the overhead time that is required for updating the processing order of the candidate pairs is comparable to the verification time, amounting to few minutes. The only exception is CF, whose verification was much larger than all other weighting schemes and has actually improved by 8.4%, as the secondary scheme breaks the ties by selecting simpler geometries.

All other patterns are insignificant.

Note that we also considered hybrid schemes by dividing CF, JS, χ^2, and ISP with the most effective weighting scheme, namely, MBRO. However, no significant improvement was observed in the respective experiments. Most importantly, hybrid schemes may deteriorate the performance of individual schemes, unlike composite ones. In D_2, the *PGR* of all hybrid schemes is reduced by ~ 80% to less than 0.100 for both Dynamic and Static Progressive GIA.nt. In D_3, there is a >35% (>54%)

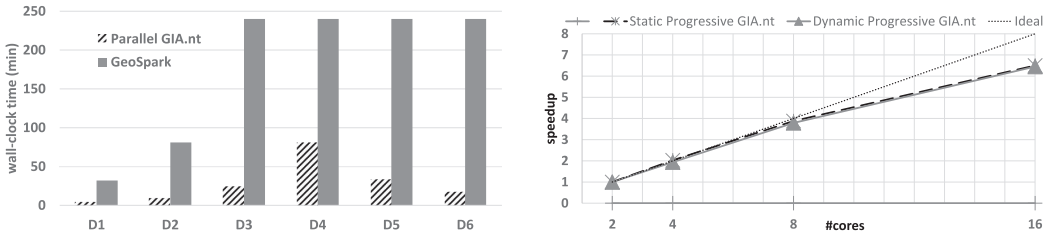

Fig. 12. The overall run-time of Parallel Batch GIA.nt in comparison with GeoSpark (on the left) and the speedup of Parallel Progressive GIA.nt as the number of cores increases over D_4, i.e., strong scalability (on the right).

decrease in *PGR* for all hybrid schemes in combination with Dynamic (Static) Progressive GIA.nt. In the best case, the performance of every hybrid scheme lies in the middle of its components: in D_4 and D_5, the *PGR* of all hybrid schemes approaches that of MBRO, without surpassing it. We omit the corresponding diagrams for brevity.

7.3 Parallel Processing

7.3.1 Batch Geospatial Interlinking. Given that RADON has not been adapted to MapReduce parallelization (only to multi-core parallelization), we compare our parallelization of batch GIA.nt with GeoSpark [35], based on the examples provided by its developers.[24] For GeoSpark, we examine only the `intersects` relation between the geometries of the datasets, because its Spark SQL API does not support the computation of the DE-9IM model. In all experiments, both systems distribute the input using the same numbers of partitions and the same Quad-Tree spatial partitioner. Figure 12 (left) reports the wall-clock time of both systems across all dataset pairs. Note that we timed out the execution of GeoSpark after 4 h of execution.

We observe that *Parallel GIA.nt consistently outperforms GeoSpark to a significant extent.* Even in the case of the dataset pair with the most complex geometries (i.e., D_4), Parallel GIA.nt is able to complete all calculations in approximately 1.5 h, while GeoSpark is still running after 4 h. Regarding D_1 and D_2, which are processed by GeoSpark within the time limit, Parallel GIA.nt requires approximately 15% of the time needed by GeoSpark. The reason is that GeoSpark uses spatial partitioning, but does not employ any tiling technique to avoid some of the redundant verifications. As a result, it verifies all the geometry pairs that coexist in the same partition.

Finally, it is worth juxtaposing Parallel GIA.nt with its serial counterpart. Comparing their overall (Wall-clock) run-time in Figure 12 (left) and Table 5, we observe *that the former is consistently faster than the latter by a whole order of magnitude.* Another advantage of Parallel GIA.nt is that it scales well to the largest dataset pair, D_6.

7.3.2 Progressive Geospatial Interlinking. Table 9 reports the performance of all methods over D_6 for two different budgets, BU = 5M and BU = 10M, using the parallel implementation of Static and Dynamic Progressive GIA.nt. In addition to the scheduling ($\mathbf{t_s}$) and verification (t_v) time, we report the *overhead time*, t_r, which is the aggregate time needed for all other computations, such as the initialization of Spark context, the spatial partitioning and the computation of the granularity of the tiles. The *overall wall-clock run-time* ($\mathbf{t_w}$) corresponds to the sum of these three time measurements.

Notice that both scheduling and verification are lower than the overhead time, mostly because of spatial partitioning. Spatial partitioning collects a random sample of the source geometries to

[24] See https://github.com/apache/incubator-sedona/tree/master/example for more details.

Table 9. Performance of (a) Static and (b) Dynamic Progressive GIA.nt Over D_6 for All Weighting Schemes in Comparison to the Optimal (Opt.) and the Random (Rnd.) Approach for Budgets of 5M and 10M Verifications

		BU = 5M							BU = 10M						
		Opt.	Rnd.	Progressive GIA.nt					Opt.	Rnd.	Progressive GIA.nt				
				CF	*JS*	χ^2	*MBRO*	*ISP*			*CF*	*JS*	χ^2	*MBRO*	*ISP*
D_6	PGR	0.890	0.039	0.025	0.065	0.064	**0.159**	0.014	0.948	0.068	0.040	0.110	0.112	**0.224**	0.028
	Recall	1.000	0.072	0.049	0.144	0.141	0.246	0.028	1.000	0.115	0.076	0.227	0.228	0.338	0.060
	Precis.	0.207	0.015	0.010	0.030	0.029	0.051	0.006	0.103	0.012	0.008	0.024	0.024	0.035	0.006
	t_r (min)	—	—	23.4	23.6	23.3	23.4	23.4	—	—	23.4	22.5	23.5	23.2	23.3
	t_s (min)	—	—	17.6	19.0	18.8	17.7	17.7	—	—	18.5	17.1	18.1	18.0	17.6
	t_v (min)	—	—	2.3	4.3	1.4	2.5	2.0	—	—	3.5	4.0	3.8	3.0	3.4
	t_w (min)	—	—	43.3	46.9	43.5	43.6	43.2	—	—	45.4	43.6	45.4	44.2	44.3

(a) Static Parallel Progressive GIA.nt

D_6	PGR	0.890	0.039	0.146	0.158	**0.241**	0.211	0.111	0.948	0.068	0.167	0.182	0.270	**0.295**	0.127
	Recall	1.000	0.072	0.182	0.183	0.277	0.323	0.133	1.000	0.115	0.195	0.231	0.324	0.423	0.159
	Precis.	0.207	0.015	0.038	0.038	0.057	0.067	0.028	0.103	0.012	0.020	0.024	0.034	0.044	0.016
	t_r (min)	—	—	23.4	23.6	23.4	23.4	23.4	—	—	23.3	22.5	23.5	23.2	23.8
	t_s (min)	—	—	17.6	18.6	17.8	18.1	18.2	—	—	17.5	19.4	18.4	17.5	17.8
	t_v (min)	—	—	3.5	1.5	2.2	3.0	3.1	—	—	4.2	2.5	3.1	4.2	3.4
	t_w (min)	—	—	44.5	43.7	43.4	44.5	44.7	—	—	45.0	44.8	45.0	44.9	45.0

(b) Dynamic Parallel Progressive GIA.nt

build the spatial partitioner and then redistributes all the geometries, invoking data shuffling. This procedure remains the same regardless the size of the budget and adds significant overhead to the overall execution. Moreover, the Verification step is faster than the Scheduling step, as most of the geometries in D_6 are small and simple, thus allowing for the quick computation of the intersection matrix. On the contrary, the Scheduling step needs to examine all the candidate pairs that passed the Filtering step, which comprises of millions of geometry pairs.

However, none of the algorithms manages to detect most of the qualifying pairs that exist within the budget, resulting to low PGR, Recall, and Precision. Still, Dynamic Progressive GIA.nt consistently outperforms its static counterpart, producing results twice as good in certain cases. MBRO also proves to be the most suitable weighting scheme for this dataset pair, too.

Note that there are differences in Recall and Precision between the static and dynamic algorithms, unlike their serial implementation. This is caused by the different pairs that the two algorithms verify. To compute PGR and the other metrics in the parallel implementation, we need to serialize the order of the discovery of the qualifying pairs. To this end, we perform the verifications in parallel and then serialize their results to get the global processing order among all partitions. Given that each partition is allocated a local budget based on the number of its source entities, the overall sum of the local budgets is a mere approximation that is typically larger than the initial budget. Yet, PGR is derived from the top *BU* pairs, which means that not all the pairs inside the priority queues of the partitions are taken into account. Due to its adaptive scheduling, Dynamic Progressive GIA.nt manages to prioritize different pairs that are more likely to be qualifying than those selected by Static Progressive GIA.nt, thus leading to better results not only in terms of PGR but also of Recall and Precision.

7.3.3 Scalability Analysis. Figure 12 (right) displays the strong scaling experiments of the parallel implementation of the progressive algorithms. In strong scaling, we examine how the overall computational time of the job scales as we increase the number of available processing units. In this experiment, the job is defined by the performance of progressive algorithms in combination with MBRO weights and *BU* = 20M, over the most time-consuming dataset pair, D_4. We observe that both algorithms scale sub-linearly and close to the ideal speedup, and we can only notice a small deceleration in the case of 16 processing units. This is because both algorithms invoke data shuffling in certain points to redistribute the geometries (i.e., spatial partitioning) and to compute the

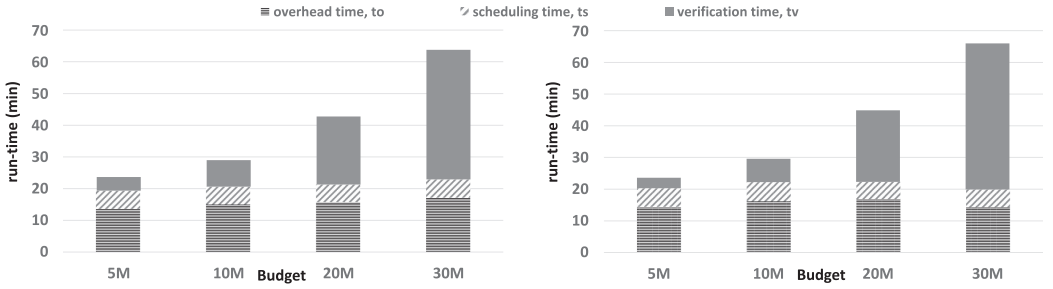

Fig. 13. Scalability of Static (left) and Dynamic (right) Parallel Progressive GIA.nt over D_4.

granularity of the tiles. By increasing the number of the processing units, more extensive data shuffling is performed, adding the extra overhead to the execution that leads to a sub-linear speedup. Overall, the total wall-clock time of Static (Dynamic) Parallel Progressive GIA.nt is reduced from 335.2 (337.5) min for 2 cores to 51.6 (52.3) min for 16 cores.

In Figure 13, we report the scalability of Static and Dynamic Parallel Progressive GIA.nt by keeping the same number of the processing units, while increasing the size of the job, i.e., by gradually increasing their budgets: $BU \in \{5M, 10M, 20M, 30M\}$. Every algorithm is combined with MBRO weights and applied to D_4. The overall wall-clock time is divided into the scheduling (t_s), the verification (t_v) and the overhead (t_r) time. We observe that both t_s and t_r remain constant, while t_v increases in proportion of the size of the budget. This is because the computations included in t_r are irrelevant to the budget size, and budgets of such sizes are not enough to significantly affect the performance of the Scheduling step. For D_4, both algorithms use around 2,000 partitions with small local budgets and, hence, with small local priority queues. As a result, the time needed to push/pop elements from these queues is negligible and does not have a significant impact to the overall performance of the Scheduling step. On the contrary, the size of the budget is decisive for the duration of the Verification step. It is worth noting that both algorithms perform quite similarly, especially with respect to t_r and t_s, but we can notice that the Verification step of the Dynamic Progressive GIA.nt is slightly slower, especially in BU = 30M, due to the continuous update of the processing order.

7.4 Discussion

We now summarize the main findings of the experiments presented above.

Starting with the batch algorithm, we observe that the main improvement of GIA.nt over RADON/RADON2 is that the memory footprint is significantly reduced by more than 66%. The larger the difference in the size of the two input datasets is, the more memory is saved by GIA.nt. This improvement, which enables GIA.nt to process much larger datasets than RADON, is achieved through implementation optimizations and by reading the target dataset from the disk, instead of loading it into main memory. The latter change, though, has no impact on the run-time, as GIA.nt remains slightly faster than RADON. Most importantly, GIA.nt has two qualitative advantages over RADON: (i) as shown in Table 6, its Filtering reduces the number of candidate pairs, while increasing their co-occurrence patterns, which are crucial for progressive methods, and (ii) it is amenable to massive parallelization, unlike RADON. Another advantage of GIA.nt is that its overall run-time is quite robust with respect to the index granularity, as shown in Figure 4.

Regarding the progressive methods, we should stress that they rely on the heuristics of weighting schemes, thus providing no performance guarantees. Instead, they depend on two factors: (i) the characteristics of the input data (e.g., the higher the proportion of qualifying pairs over the

candidate ones, the better the performance), and (ii) the size of the budget. Regarding the latter, there is a clear trade-off between recall and precision in Tables 7, 8 and 9: larger budgets increase recall at the cost of lower precision and vice versa, for smaller budgets.

To assess the performance of progressive methods, we compared them two baseline methods: the Optimal Scheduling, which places all qualifying pairs before the non-qualifying ones, and the Random Scheduling, which produces an arbitrary ordering of all candidate pairs. The closer the progressive methods are to the former baseline method, the better. Our experimental results demonstrate that even though there is still room for improving their performance, they significantly outperform the latter baseline. In fact, *MBRO* consistently outperforms Random Scheduling in combination with Static Progressive GIA.nt, as shown in Table 7.

The same applies to the rest of the weighting schemes when they are combined with Dynamic Progressive GIA.nt, especially when they form composite schemes with *MBRO* as the secondary one. In Tables 8 and 9(b), we observe that Random Scheduling outperforms only *ISP* in D_4 for BU = 10M as well as *CF* in D_1 for BU = 5M and in D_3 for both budgets. In some other cases, the differences seem negligible, but when considering the performance of Optimal Scheduling, they become significant. For instance, in D_2, the precision of Random Scheduling is 0.013 and of *ISP* just 0.014 for BU = 10M; considering, though, that the precision of Optimal Scheduling is 0.020, the difference between the two approaches can be characterized as significant.

Regarding time efficiency, we observe in Table 5 that the filtering time, t_f, accounts for a negligible portion of the overall run-time, which is dominated by the verification time, t_v. In Tables 7, 8, and 9, we notice that the scheduling time, t_s, is much higher than t_f, albeit significantly lower than t_v in most cases. In general, the scheduling time is significant with respect to t_v, if (i) the budget is very low, yielding very low t_v, (ii) the selected geometries are very small and simple, as in the case of D_5, or (iii) the number of candidate pairs with intersecting MBRs is very high. *The more candidate pairs have intersecting MBRs, the more time consuming is Scheduling*, because its time complexity depends linearly on their number, as shown in Table 3. Note that t_s is not affected by the tiles instersecting the MBR of each target geometry, because their contents, i.e., the source geometry ids they contain, are efficiently added to the current set of candidate pairs and those with disjoint MBRs are later discarded. Most importantly, though, t_s involves the time required to read the target geometries from the disk, unlike t_v, since progressive verification merely processes the top-weighted target geometries, which have already been stored in main memory during Scheduling.

In any case, though, the cost of Scheduling should be assessed with respect to the batch verification time. Comparing t_s in Table 7 with t_v in Table 5, we observe that the former consistently amounts to less than 5% of the latter. Even in the case of D_5, t_s+t_v actually corresponds to 2.5% of batch t_v, which means that Static Progressive GIA.nt in combination with *MBRO* spends less than 2.5% of the batch verification time to detect 10M/164M = 6.1% of the qualifying pairs. The same applies to *JS*, χ^2 and *ISP*, which also achieve very high recall for BU = 10M over D_5. In the other datasets, this trade-off is even better. E.g., in D_3 for BU = 10M, *JS* and χ^2 detect 83% of all qualifying pairs within 10% of the batch verification time, while χ^2 and *ISP* detect 76% of all qualifying pairs for the same portion of the batch verification time.

In the case of Dynamic Progressive GIA.nt, the overhead time t_o, which is required for weight updating, is rather low when comparing its value in Figure 10(ii) to the corresponding verification time in Table 8 (seconds or minutes in comparison to hours). The reason is that Algorithm 3 reranks only those pairs among the top-BU weighted that have not been verified so far. In fact, very few pairs are re-weighted whenever a new qualifying pair is detected.

Regarding the relative run-time of the five weighting schemes, we observe that *CF* consistently yields the highest verification time, followed by *MBRO*. All other schemes are much faster, yielding

similar verification times, with *ISP* being consistently the fastest one. This is explained by the diagrams of Figure 9, which report the complexity of the top-weighted candidate pairs in terms of their area as well as the number of their boundary points. The smaller both measures are for both source and target geometries, the faster is the corresponding progressive verification time. Regarding the scheduling time, though, it remains relatively stable across all weighting schemes, as it is dominated by the time required to read the target geometries from the disk.

Considering the relative effectiveness of the weighting schemes, we observe that *CF* exhibits the lowest performance in most cases. This outlier behavior is explained in Figure 8, which reveals that *CF* typically produces scores of very low distinctiveness and has a limited scope, as its search space is reduced to geometries with large MBRs. For this reason, its combination with Dynamic Progressive GIA.nt yields a lower performance, as shown in Figure 10, unlike all other weighting schemes. In fact, the characteristics of the other weighting schemes are highly correlated, yielding similarly high source and target diversities in most cases. Among them, *MBRO* excels in distinctiveness, thus being ideal for the secondary role in a composite scheme. In this way, it manages to boost the performance of all other weighting schemes, even *CF*, in the context of Dynamic Progressive GIA.nt, as shown in Table 8.

Finally, it is worth stressing that our massive parallelization scheme significantly improves the run-time of all algorithms, i.e., the batch GIA.nt and its progressive variants.

8 CONCLUSIONS

In this article, we defined Holistic Geospatial Interlinking as the task of simultaneously computing all non-trivial DE-9IM topological relations between the input geometries. To solve it, we proposed GIA.nt, an algorithm that provides several advantages over the state-of-the-art approach RADON—both quantitative (significantly lower space requirements for slightly lower running times) and qualitative (easier massive parallelization and stronger co-occurrence patterns for progressive approaches). 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 considered two progressive algorithms: (i) Static Progressive GIA.nt, which produces an immutable processing order of geometry pairs, and (ii) Dynamic Progressive GIA.nt, which updates the processing order on-the-fly, based on the qualifying pairs that are detected. Both algorithms are equipped with five weighting schemes, which a-priori estimate the likelihood that a pair of geometries satisfies at least one topological relation. A series of experiments over several real dataset pairs verified that Dynamic Progressive GIA.nt consistently achieves the top performance, especially when combined with composite weighting schemes, where the secondary one resolves the ties of the primary one. Depending on the input data, the following weighting schemes should be preferred:

(1) If the two input datasets are heterogeneous, i.e., the one contains LineStrings and the other Polygons, then *MBRO* achieves the highest performance, on average (see D_1, D_2, D_4 in Table 8 and D_6 in Table 9). That is, *MBRO* is the best choice if there is no background knowledge about the type of topological relations to be discovered. However, if the majority of the qualifying pairs satisfies the relation `touches` (as in D_2), then *JS+MBRO* outperforms all weighting schemes, whereas a high proportion of the relations `contains`, `covered-by`, or `covers` favors *CF+MBRO* (as in D_4).
(2) In the case of homogeneous input data with Polygons (as D_3), *MBRO* clearly constitutes the most effective weighting scheme. Yet, *JS+MBRO* and χ^2*+MBRO* offer equivalent performance for slightly lower verification times, since they promote smaller and simpler geometries.

(3) In the case of homogeneous input data with LineStrings (as D_5), all composite weighting schemes achieve very high effectiveness, except *CF*. However, *ISP+MBRO* reduces the verification time to a significant extent, due to its emphasis on low complexity pairs, at the cost of slightly lower effectiveness.

If possible, then these configurations should be run in parallel, on top of Apache Spark, which is able to reduce the overall run-time by a whole order of magnitude.

In the future, we will focus on the Verification step, trying to minimize the time required for computing the Intersection Matrix for each pair of geometries. We will also consider 3D geometries as input, with time constituting the third dimension. Finally, we intend to explore the use of supervised progressive approaches.

REFERENCES

[1] Abdullah Fathi Ahmed, Mohamed Ahmed Sherif, and Axel-Cyrille Ngonga Ngomo. 2018. RADON2—A buffered-intersection matrix computing approach to accelerate link discovery over geo-spatial RDF knowledge bases: OAEI2018 results. In *Proceedings of the International Workshop on Ontology Matching*. 197–204.

[2] Edward P. F. Chan and Jimmy N. H. Ng. 1997. A general and efficient implementation of geometric operators and predicates. In *Proceedings of the International Symposium on Advances in Spatial Databases (SSD'97)*, Vol. 1262. 69–93.

[3] Eliseo Clementini, Paolino Di Felice, and Peter van Oosterom. 1993. A small set of formal topological relationships suitable for end-user interaction. In *Proceedings of the International Symposium on Advances in Spatial Databases (SSD'93)*. 277–295.

[4] Eliseo Clementini, Jayant Sharma, and Max J. Egenhofer. 1994. Modelling topological spatial relations: Strategies for query processing. *Comput. Graph.* 18, 6 (1994), 815–822.

[5] Isabel F. Cruz, Flavio Palandri Antonelli, and Cosmin Stroe. 2009. AgreementMaker: Efficient matching for large real-world schemas and ontologies. *Proc. VLDB Endow.* 2, 2 (2009), 1586–1589.

[6] Jens-Peter Dittrich and Bernhard Seeger. 2000. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'00)*. 535–546.

[7] Max J. Egenhofer and Robert D. Franzosa. 1991. Point-set topological spatial relations. *Int. J. Geogr. Info. Syst.* 5, 2 (1991), 161–174.

[8] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'15)*. IEEE Computer Society, 1352–1363.

[9] Theofilos Ioannidis, George Garbis, Kostis Kyzirakos, Konstantina Bereta, and Manolis Koubarakis. 2021. Evaluating geospatial RDF stores using the benchmark geographica 2. *J. Data Semant.* 10, 3–4 (2021), 189–228.

[10] Mahmoud Ismail, Ermias Gebremeskel, Theofilos Kakantousis, Gautier Berthou, and Jim Dowling. 2017. Hopsworks: Improving user experience and development on hadoop with scalable, strongly consistent metadata. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'17)*. IEEE Computer Society, 2525–2528.

[11] Edwin H. Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Trans. Database Syst.* 32, 1 (2007), 7.

[12] Krzysztof Janowicz, Yingjie Hu, Grant McKenzie, Song Gao, Blake Regalia, Gengchen Mai, Rui Zhu, Benjamin Adams, and Kerry L. Taylor. 2016. Moon landing or safari? A study of systematic errors and their causes in geographic linked data. In *Proceedings of the International Conference on Geographic Information Science (GIScience'16)*. 275–290.

[13] Krzysztof Janowicz, Simon Scheider, Todd Pehle, and Glen Hart. 2012. Geospatial semantics and linked spatiotemporal data—Past, present, and future. *Semantic Web* 3, 4 (2012), 321–332.

[14] Oje Kwon and Ki-Joune Li. 2011. Progressive spatial join for polygon data stream. In *Proceedings of ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL'11)*. 389–392.

[15] George Mandilaras, Despina-Athanasia Pantazi, Manolis Koubarakis, Nick Hughes, Alistair Everett, and Åshild Kiærbech. 2020. Ice Monitoring With ExtremeEarth. In *Workshop on Large Scale RDF Analytics LASCAR II, collocated with ESWC'20*. http://www.earthanalytics.eu/publications/ExtremeEarthIceDemo.pdf.

[16] Amin Mobasheri. 2017. A rule-based spatial reasoning approach for OpenStreetMap data quality enrichment; case study of routing and navigation. *Sensors* 17, 11 (2017), 2498.

[17] Axel-Cyrille Ngonga Ngomo. 2013. ORCHID—Reduction-ratio-optimal computation of geo-spatial distances for link discovery. In *Proceedings of the International Semantic Web Conference (ISWC'13)*. 395–410.

[18] Salman Niazi, Mahmoud Ismail, Seif Haridi, and Jim Dowling. 2019. HopsFS: Scaling hierarchical file system metadata using NewSQL databases. In *Encyclopedia of Big Data Technologies*. Springer.

[19] George Papadakis, Georgios Mandilaras, Nikos Mamoulis, and Manolis Koubarakis. 2021. Progressive, holistic geospatial interlinking. In *Proceedings of the Web Conference*.

[20] Thorsten Papenbrock, Arvid Heise, and Felix Naumann. 2015. Progressive duplicate detection. *IEEE Trans. Knowl. Data Eng.* 27, 5 (2015), 1316–1329.
[21] Jignesh M. Patel and David J. DeWitt. 1996. Partition based spatial-merge join. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 259–270.
[22] Matthew Perry and John Herring. 2012. OGC GeoSPARQL-A geographic query language for RDF data. *OGC Implement. Stand.* 40 (2012).
[23] Blake Regalia, Krzysztof Janowicz, and Grant McKenzie. 2019. Computing and querying strict, approximate, and metrically refined topological relations in linked geographic data. *Trans. GIS* 23, 3 (2019), 601–619.
[24] Georgios M. Santipantakis, Christos Doulkeridis, Akrivi Vlachou, and George A. Vouros. 2020. Integrating data by discovering topological and proximity relations among spatiotemporal entities. In *Big Data Analytics for Time-critical Mobility Forecasting: From Raw Data to Trajectory-oriented Mobility Analytics in the Aviation and Maritime Domains*. Springer, 155–179.
[25] Georgios M. Santipantakis, Apostolos Glenis, Christos Doulkeridis, Akrivi Vlachou, and George A. Vouros. 2019. stLD: Towards a spatio-temporal link discovery framework. In *Proceedings of the International Workshop on Semantic Big Data (SBD@SIGMOD'19)*. 4:1–4:6.
[26] Tzanina Saveta, Irini Fundulaki, Giorgos Flouris, and Axel-Cyrille Ngonga Ngomo. 2018. SPgen: A benchmark generator for spatial link discovery tools. In *Proceedings of the International Semantic Web Conference (ISWC'18)*. 408–423.
[27] Mohamed Ahmed Sherif, Kevin Dreßler, Panayiotis Smeros, and Axel-Cyrille Ngonga Ngomo. 2017. RADON—Rapid discovery of topological relations. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 175–181.
[28] Giovanni Simonini, Sonia Bergamaschi, and H. V. Jagadish. 2016. BLAST: A loosely schema-aware meta-blocking approach for entity resolution. *Proc. VLDB Endow.* 9, 12 (2016), 1173–1184.
[29] Giovanni Simonini, George Papadakis, Themis Palpanas, and Sonia Bergamaschi. 2019. Schema-agnostic progressive entity resolution. *IEEE Trans. Knowl. Data Eng.* 31, 6 (2019), 1208–1221.
[30] Panayiotis Smeros and Manolis Koubarakis. 2016. Discovering spatial and temporal links among RDF data. In *Proceedings of the Workshop on Linked Data on the Web (LDOW'16), Co-located with 25th International World Wide Web Conference (WWW'16)*.
[31] Wee Hyong Tok, Stéphane Bressan, and Mong-Li Lee. 2006. Progressive spatial join. In *Proceedings of the 18th International Conference on Scientific and Statistical Database Management (SSDBM'06)*. 353–358.
[32] Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2019. Parallel in-memory evaluation of spatial joins. In *Proceedings of the International Conference on Advances in Geographic Information Systems (SIGSPATIAL'19)*. 516–519.
[33] Steven Euijong Whang, David Marmaros, and Hector Garcia-Molina. 2013. Pay-as-you-go entity resolution. *IEEE Trans. Knowl. Data Eng.* 25, 5 (2013), 1111–1124.
[34] Zhengcong Yin, Chong Zhang, Daniel W. Goldberg, and Sathya Prasad. 2019. An NLP-based question answering framework for spatio-temporal analysis and visualization. In *Proceedings of the 2nd International Conference on Geoinformatics and Data Analysis*. 61–65.
[35] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. 2019. Spatial data management in apache spark: The GeoSpark perspective and beyond. *GeoInformatica* 23, 1 (2019), 37–78.

Received April 2021; revised September 2021; accepted December 2021