Web Semantics: Science, Services and Agents on the World Wide Web 81 (2024) 100817

Contents lists available at ScienceDirect

Web Semantics: Science, Services and Agents on the World Wide Web

journal homepage: www.elsevier.com/locate/websem

Three-dimensional Geospatial Interlinking with JedAI-spatial

Marios Papamichalopoulos [a], George Papadakis [a,*], George Mandilaras [a], Maria Siampou [a], Nikos Mamoulis [b], Manolis Koubarakis [a]

[a] National and Kapodistrian University of Athens, Greece
[b] University of Ioannina, Greece

ARTICLE INFO

Dataset link: https://zenodo.org/record/6384164, https://github.com/AI-team-UoA/JedAI-spatial

Keywords:
Link Discovery
Geospatial Interlinking
DE-9IM topological relations
Batch & progressive methods
Parallelization

ABSTRACT

Geospatial data constitutes a considerable part of Semantic Web data, but so far, its sources are inadequately interlinked in the Linked Open Data cloud. Geospatial Interlinking aims to cover this gap by associating geometries with topological relations like those of the Dimensionally Extended 9-Intersection Model. Due to its quadratic time complexity, various algorithms aim to carry out Geospatial Interlinking efficiently. We present *JedAI-spatial*, a novel, open-source system that organizes these algorithms according to three dimensions: (i) *Space Tiling*, which determines the approach that reduces the search space, (ii) *Budget-awareness*, which distinguishes interlinking algorithms into batch and progressive ones, and (iii) *Execution mode*, which discerns between serial algorithms, running on a single CPU-core, and parallel ones, running on top of Apache Spark. We analytically describe JedAI-spatial's architecture and capabilities and perform thorough experiments to provide interesting insights about the relative performance of its algorithms.

1. Introduction

Geospatial data has escalated tremendously over the years. The outbreak of Internet of Things (IoT) devices, smartphones, position tracking applications and location-based services has skyrocketed the volume of geospatial data. For example, 100TB of weather-related data is produced everyday[1]; Uber hit the milestone of 5 billion rides among 76 countries already on May 20, 2017.[2] Web platforms like OpenStreetMap[3] provide an open and editable map of the whole world. Earth observation programmes like Copernicus[4] publish tens of terabytes of geospatial data per day on the Web.[5] For these reasons, geospatial data constitutes a considerable part of Web data, but the links between its data sources and their geometries are scarce in the Linked Open Data cloud [1,2].

Geospatial Interlinking aims to cover this gap by associating pairs of geometries with topological relations like those of the Dimensionally Extended 9-Intersection Model (DE-9IM) [3–5]. As an example consider Fig. 1, where LineString g_3 `intersects` LineString g_4 and `touches` Polygon g_1, which `contains` Polygon g_2. Two are the main challenges of this task: (i) its inherently quadratic time complexity, because it has to examine every pair of geometries, and (ii) the high time complexity of examining a single pair of geometries, which amounts to $O(N \log N)$, where N is the size of the union set of their boundary points [6]. As a result, Geospatial Interlinking involves a high computational cost that does not scale to large Web datasets.

Numerous algorithms aim to address these challenges by enhancing the time efficiency and scalability of Geospatial Interlinking. The most recent ones operate in main memory, reducing the search space to pairs of geometries that are likely to be topologically related according to a geospatial index [7–9]. However, no open-source system organizes these algorithms into a common framework that facilitates researchers and practitioners in their effort to populate the LOD cloud with more topological relations. Systems like Silk [10] and LIMES [11] convey only the methods developed by their creators, Silk-spatial [12] and RADON [13] respectively. Systems like stLD [14,15] could act as a library of established methods, but are not publicly available. Moreover, no system supports progressive methods, which produce results in a

* Corresponding author.
E-mail addresses: cs3190006@di.uoa.gr (M. Papamichalopoulos), gpapadis@di.uoa.gr (G. Papadakis), gmandi@di.uoa.gr (G. Mandilaras), m.siampou@di.uoa.gr (M. Siampou), nikos@cs.uoi.gr (N. Mamoulis), koubarak@di.uoa.gr (M. Koubarakis).

[1] https://www.ibm.com/topics/geospatial-data
[2] https://www.uber.com/en-SG/blog/uber-hits-5-billion-rides-milestone
[3] https://www.openstreetmap.org
[4] https://www.copernicus.eu
[5] https://www.copernicus.eu/sites/default/files/Copernicus_DIAS_Factsheet_June2018.pdf

https://doi.org/10.1016/j.websem.2024.100817
Received 11 July 2022; Received in revised form 8 September 2023; Accepted 9 March 2024
Available online 24 March 2024

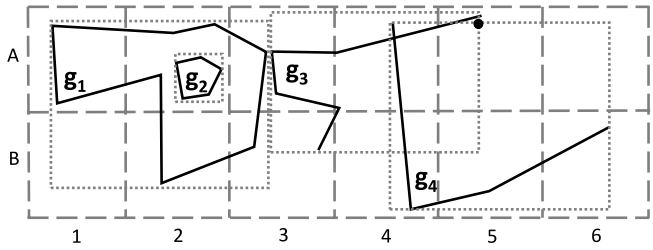

Fig. 1. Example of four topologically related geometries.

pay-as-you-go manner, thus being indispensable for applications with limited computational or temporal resources [2].

To address these issues, we present *JedAI-spatial*, an open-source system that supports a broad range of Geospatial Interlinking applications by implementing all state-of-the-art methods. JedAI-spatial makes the following contributions:

- It organizes the main algorithms into a novel taxonomy that facilitates their use according to the application requirements.
- Its intuitive user interface supports both lay and expert users.
- It facilitates the benchmarking of the state-of-the-art algorithms, while its modular and extensible architecture allows for easily incorporating new ones.
- We have publicly released all data and the code of JedAI-spatial under Apache License V2.0.[6] [16]

An extended version of this work is presented in [17].

2. Preliminaries

JedAI-spatial supports two types of geometries: (i) the one-dimensional *LineStrings*, which comprise a sequence of points and the line segments that connect the consecutive ones (e.g., g_3 and g_4 in Fig. 1), and (ii) the two-dimensional *Polygons*, which usually comprise a sequence of connected points, where the first and the last one coincide (e.g., g_1 and g_2 in Fig. 1).

Both types of geometries consist of an interior, a boundary and an exterior (i.e., all points that are not part of the interior or the boundary). These three parts are used by the DE-9IM model, which has been standardized by the Open Geospatial Consortium (OGC), to define 10 topological relations between two geometries A and B with self-explanatory names: `equals`, `disjoint`, `intersects`, `touches`, `within`, `contains`, `covers`, `covered-by`, `crosses` and `overlaps` [17].

Note that JedAI-spatial disregards the relation `disjoint` because it provides no positive information for the relative location of two geometries, while it is impractical to compute it in the case of large input data — it scales quadratically with the input size, given that the vast majority of geometries share no interior or boundary point [2]. JedAI-spatial relies on a closed-world assumption: the lack of the relation `intersects` between two geometries implies that they are `disjoint`.

Following [2,18], JedAI-spatial considers *Holistic Geospatial Interlinking*, which simultaneously computes all **positive** topological relations (i.e., all DE-9IM relations except for `disjoint`): for each pair of geometries, it estimates the Intersection Matrix, a 3×3 matrix that contains the dimensions of the intersection between the interior, the boundary and the exterior of two geometries such that all relations can be extracted with boolean expressions.[7] This task is formally defined as:

[6] https://github.com/AI-team-UoA/JedAI-spatial
[7] https://en.wikipedia.org/wiki/DE-9IM#Matrix_model

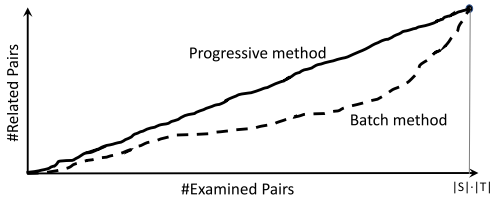

Fig. 2. Progressive Geometry Recall for batch and progressive methods.

Problem 1 (*Holistic Geospatial Interlinking*)**.** Given a source and a target dataset, S and T, together with the set of positive topological relations R, compute the set of links $L_R = \{(s,r,t) \subseteq S \times R \times T : r(s,t)\}$ from the Intersection Matrix of all topologically related geometry pairs.

Progressive Geospatial Interlinking. An *approximate* solution to Geospatial Interlinking is provided by progressive algorithms, which run for a limited time or number of calculations. These algorithms are necessary for applications with limited resources, such as cloud applications with a specific budget for AWS Lambda functions, which charge when called [19].

Compared to batch algorithms, the goal of progressive algorithms is twofold [2]: (i) they should produce the same results if they process the entire input data, and (ii) they should detect a significantly larger number of related geometry pairs, if their operation is terminated earlier.

These requirements are reflected in Fig. 2, where the horizontal axis corresponds to the number of examined pairs and the vertical one to the number of related pairs. Essentially, the progressive algorithms should define a processing order that examines the related pairs before the non-related ones, unlike batch algorithms, which examine candidate pairs in an arbitrary order. Hence, the progressive algorithms should maximize the area under their curve, an evaluation measure that is captured by *Progressive Geometry Recall* (**PGR**) and is defined in $[0,1]$, with higher values indicating higher effectiveness. More formally, progressive algorithms tackle the following task [2]:

Problem 2 (*Progressive Geospatial Interlinking*)**.** Given a source and a target dataset, S and T, the positive topological relations R and a budget BU, maximize PGR@BU.

3. System architecture

JedAI-spatial organizes the Geospatial Interlinking algorithms into a novel taxonomy formed by three dimensions:

(1) *Space Tiling* distinguishes the algorithms into *grid-*, *tree-* and *partition-based* ones. The first type includes Semantic Web techniques that define a static or dynamic Equigrid, the second one encompasses main-memory spatial join techniques from the database community, and the third one conveys variations of plane sweep, a cornerstone of computational geometry.

(2) *Budget-awareness* categorizes algorithms into *batch* and *progressive* ones. The former are executed in a budget-agnostic manner that processes the input data in no particular order and produces results only upon completion of the entire process. Progressive algorithms are suitable for applications with limited computational or temporal resources, producing results in a budget-aware, pay-as-you-go manner.

(3) *Execution mode* distinguishes between serialized algorithms, which run on a single CPU core, and massively parallel ones, which run on Apache Spark.

JedAI-spatial creates end-to-end pipelines that are defined by these three dimensions. This is achieved by the architecture in Fig. 3: JS-gui offers two interfaces for user interaction (view), JS-core conveys numerous algorithms and pipelines (controller), and the Data Model

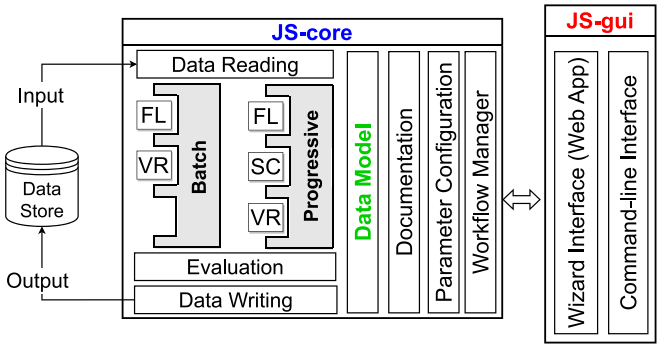

Fig. 3. The model-view-controller architecture of JedAI-spatial.

component provides the data structures that lie at its core (model). This architecture serves the following goals:

• *Broad data coverage.* Through its *Data Reading* component, JedAI-spatial supports the most popular structured and semi-structured formats that are used for encoding geometries: Well Known Text (WKT), GeoJSON, CSV and TSV files, JsonRDF as well as GeoSPARQL vocabulary, accessed through RDF dumps or SPARQL endpoints. In this way, JedAI-spatial is able to interlink heterogeneous datasets, e.g., WKT with GeoJSON.

• *Broad algorithmic coverage.* JedAI-spatial serves as a library of the state-of-the-art algorithms in the literature, even if they have not been applied to Geospatial Interlinking before. This applies to spatial join algorithms, which are adapted to detect topological relations for the first time, as explained in Section 7.

• *Broad application coverage.* JedAI-spatial accommodates both academic and commercial applications, as its code is released under Apache License V2.0. It also supports both batch and progressive applications. In any type of applications, it is crucial to detect the most suitable algorithm for the data at hand (e.g., different algorithms might excel in a LineString-to-LineString scenario than in a Polygon-to-LineString one). To cover this need, JedAI-spatial's benchmarking functionality facilitates the comparative evaluation of a large variety of pipelines.

• *High usability.* JedAI-spatial supports both novice and expert users. The former can apply complex pipelines to their data simply by choosing among the available algorithms, without any knowledge about their internal functionality or their configuration (see Section 5.2). Power users can use JedAI-spatial as a library or a Maven dependency, can manually fine-tune the selected methods and can extend it with more algorithms or pipelines according to their needs.

• *Extensibility.* Every algorithm in JedAI-spatial implements the interface of its workflow step, which determines its input and output. New methods can be seamlessly integrated into JedAI-spatial by implementing the respective interface so that they are treated like the existing ones. New workflow steps can also be added as long as they define a new interface specifying their input and output. All additions should implement the `IDocumentation` interface (see Section 5.1).

• *Efficiency and scalability.* JedAI-spatial scales well to large datasets both in stand-alone systems (cf. Section 4.1) and in Spark clusters (cf. Section 4.2).

4. Back-end: JS-core

All methods have been re-implemented in JedAI-spatial's common framework, thus minimizing the dependencies to other systems and libraries. For most algorithms, we have incorporated improvements that significantly enhance their original performance — see [17] for more details.

4.1. Serial algorithms

Following all relevant open-source libraries in the literature (i.e., Silk and LIMES), the serial algorithms of JS-core are implemented in Java, which facilitates the deployment of our system (due to its portability), its use as a library (through Maven), its extension by practitioners and researchers (through the public interfaces) as well as its maintenance (due to its object-oriented capabilities).

4.1.1. Batch algorithms

The methods of this category address Problem 1. JedAI-spatial implements the state-of-the-art ones according to extensive experimental analyses [8,9,20]. To compute all positive topological relations between the source and the target geometries, they follow a two-step pipeline: initially, **the Filtering step** indexes the source dataset and, if necessary, the target one, based on the minimum bounding rectangle (MBR) of each geometry — in Fig. 1, the MBRs are the dotted rectangles surrounding each geometry. The resulting index is used to generate C, the set of candidate pairs, which are likely to satisfy at least one topological relation. Next, **the Verification step** examines every pair in C as long as their MBRs are intersecting. The detected topological relations are added to the set of triples L, i.e., the output. JedAI-spatial organizes these algorithms into three subcategories, based on the type of the index used in Filtering i.e., according to the Space Tiling dimension:

(1) Grid-based Algorithms. The input geometries are indexed by dividing the Earth's surface into cells of the same dimensions. The index is called *Equigrid* and its cells *tiles*. Every geometry is placed into the tiles that intersect its MBR. JedAI-spatial conveys four state-of-the-art algorithms of this type, which differ in the definition and use of the Equigrid during Filtering and Verification.

• RADON [13]. Filtering loads both input datasets into main memory and defines an Equigrid index by setting the horizontal and vertical dimensions of its tiles equal to the average width and height, respectively, over all geometries. Verification computes the Intersection Matrix for all candidate pairs [18], taking special care to avoid the ones repeated across different tiles.

• GIA.nt [2]. Filtering loads in memory the input dataset with the fewest geometries. The granularity of the Equigrid index is determined by the average dimensions of this dataset. Verification reads the geometries of the other dataset from the disk. For each geometry g, it sets as candidates those with an MBR intersecting the same tiles as $MBR(g)$. Then, it computes their Intersection Matrix, adding the detected links to L.

• Static variants. Unlike the *dynamic* Equigrid of the above algorithms, *Silk-spatial* [12] employs a *static* Equigrid, whose granularity is predetermined (by the user), independently of the input data. The resulting index might be too fine- or coarse-grained for the input datasets, but the candidate pairs are eventually filtered out if their MBRs are disjoint. To put this approach into practice, JedAI-spatial includes the custom methods Static RADON and Static GIA.nt.

(2) Partition-based Algorithms. They rely on a (usually vertical) sweep line that moves across the Earth's surface, stopping at some points. Filtering sorts all input geometries in ascending order of their lower boundary on the horizontal axis, x_{min}. Verification is restricted to pairs of source and target geometries whose MBRs simultaneously intersect the sweep line at each stop. It terminates once the sweep line processes all geometries.

• Plane Sweep [21]. It applies the above process to S and T. Before verifying a pair of geometries, it ensures that they overlap on the y-axis.

• PBSM [22]. It splits the given geometries into a manually defined number of orthogonal partitions and applies Plane Sweep inside every partition. Filtering defines the partitions, assigns every geometry to all partitions that intersect its MBR and sorts all geometries per partition in ascending x_{min}. Verification goes through the partitions and in each of them, it sweeps a vertical line l, computing the Intersection Matrix for

each pair of geometries that simultaneously intersect l and overlap on the y-axis. It avoids repeated verifications of the same geometry pairs across different partitions through the *reference point technique*, which verifies two geometries only in the partition containing the top-left corner of the intersection of their MBRs [23].

• Stripe Sweep. To lower the time complexity of Plane Sweep, this *new* algorithm sorts only the geometries of the smallest input dataset during Filtering, S. These geometries are then partitioned into several vertical stripes, whose length is equal to the average width of the source geometries. Every source geometry $s \in S$ is placed in all stripes that intersect its MBR. Verification aggregates the *set* of source geometries contained in the stripes intersecting each $t \in T$. This set is further refined by retaining only the candidates with intersecting MBRs.

To clarify the difference between these algorithms, it is worth explaining the data structures that lie at their core. JedAI-spatial equips Plane Sweep and PBSM with two different sweep structures for maintaining the active geometries, whose MBR intersects the sweep-line in its current position:

(i) *List Sweep* maintains one linked list for each input dataset. In every move of the sweep line l, the contents of both lists are updated, inserting the geometries with an intersecting MBR and removing the expired ones, i.e., the geometries with $x_{max} < l_x$.

(ii) *Striped Sweep* splits the given datasets into n stripes and uses a different List Sweep per stripe. After preliminary experiments, the length of each stripe on the horizontal axis was set to the average width of the source geometries.

Stripe Sweep can use two different data structures for storing the source geometries per stripe: (i) a hash map, which associates every stripe id with the corresponding source geometry ids, and (ii) an STR-Tree [24], which indexes the source geometries in each stripe. The hash map does not ensure the overlap on the y-axis before checking the MBR intersection of candidate pairs, unlike the STR-Tree.

(3) Tree-based Algorithms. These algorithms rely on state-of-the-art spatial tree indices. During Filtering, they index the smallest input dataset. During Verification, every geometry g from the other dataset queries the tree index; its candidates are located in the leaf nodes whose MBR intersects with $MBR(g)$. For all candidate geometries with an MBR that intersects $MBR(g)$, the Intersection Matrix is computed.

• R-Tree [25]. In this index, every non-leaf node contains pointers to its child nodes along with an MBR that encloses the span of all the MBRs in its children. Every leaf node contains up to M geometries. When an entry is added to a full node, the node is split into two new ones, which are initialized with the two largest geometries. Each of the remaining geometries is added to the node whose MBR expands the least after insertion.

• Quadtree [26]. In this index, every non-leaf node has exactly four children, dividing the space into four quadrants: north-east, north-west, south-east and south-west. Again, every node has a maximum capacity M. When M is reached, the corresponding cell is split into four new children.

• CR-Tree [27]. This index compresses the R-Tree so that it leverages the L1 and L2 cache memory of CPUs, which have faster access times. The *Quantized Relative Representation of MBR* minimizes the size of the MBRs, which dominate the space requirements. CR-Trees are usually wider and more shallow than R-Trees, due to their higher branching factor, achieving higher time efficiency and occupying ~60% less memory.

4.1.2. Progressive algorithms

This category encompasses methods that address Problem 2. Their goal is to maximize the number of related geometry pairs that are detected after consuming the available budget BU, which determines the maximum number of verifications. JedAI-spatial implements the state-of-the-art ones [2,28], which follow a three-step pipeline: Filtering is identical with that of batch methods, producing a set of candidate pairs C. **Scheduling** first refines C by discarding the pairs with non-overlapping MBRs. Then, it defines the processing order of the remaining pairs so that the likely related ones are placed before the unlikely ones. The new set of candidate pairs C' is forwarded to Verification, which carries out their processing and returns the set of detected links, L.

The gist of progressive algorithms is the combination of Scheduling with Filtering, as Verification remains the same in all cases. Based on the co-occurrence frequency of geometries in the tiles of *grid-based* Filtering, Scheduling assigns a score to every pair of candidates with intersecting MBRs (note that the tree- and partition-based algorithms detect the co-occurrence of geometry pairs, without the corresponding frequency). The higher this score is, the more likely are the constituent geometries to satisfy at least one topological relation. JedAI-spatial offers all weighting schemes defined in [2,28]. These are leveraged by the following algorithms:

• Progressive GIA.nt [2]. It applies the same Filtering as its batch GIA.nt. Its Scheduling gathers in a priority queue the top-BU weighted candidate pairs.

• Dynamic Progressive GIA.nt [28]. It uses the same Filtering and Scheduling as Progressive GIA.nt. Its Verification does not employ a static processing order, but updates the processing order of the top-weighted candidate pairs dynamically, as more topologically related pairs are detected: whenever a pair of geometries (s, t) is detected as topologically related, it updates the weight w of all top-ranked candidate pairs that include s or t, but have not been processed yet through the following formula: $w' = w \times (1 + q)$, where q is the number of times a geometry of this candidate pair has been verified as topologically related. Promoting the weight of other pairs in this way is useful in cases where one dataset involves long LineString geometries like roads or rivers, whereas the other involves Polygon geometries like buildings or cities: 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.

• Progressive RADON [2]. It applies RADON's Filtering and defines the processing order of the resulting tiles by sorting them in increasing or decreasing number of candidate pairs, a hyperparameter that depends on the data at hand. Inside every tile, it removes the redundant candidates with the reference point technique. The rest are processed in decreasing score, as determined by the selected weighting scheme. Thus, the pairs most likely to satisfy topological relations are processed first inside every tile. This is a local approximation of the global sorting used by Progressive GIA.nt.

Note that three more progressive algorithms are presented in [17]: Local Progressive GIA.nt, Geometry-ordered GIA.nt and Iterative Progressive GIA.nt. We omit them for brevity, as they have not achieved high performance in practice.

4.2. Parallel algorithms

To scale to voluminous datasets, JedAI-spatial exploits the massive parallelization functionalities offered by Apache Spark. JedAI-spatial has aggregated all relevant algorithms in the literature that are crafted for the same framework and the same types of geometries, i.e., LineStrings and Polygons [7] (we exclude SIMBA [29], which exclusively applies to points). Note that these algorithms leverage grid or tree indices, but partition-based parallel approaches are also possible.

We adapted all algorithms to the pipeline in Fig. 4 so as to reduce the time-consuming Spark shuffles, increasing the overall performance. The pipeline comprises three steps: (i) The *Preprocessing Stage* reads the source and target datasets from HDFS, transforms them into Spark RDDs and partitions them according to a predetermined approach. (ii) The *Global Join Stage* joins the source and target partitions that are overlapping and assigns every pair of overlapping partitions to a different worker for processing. (iii) In the *Local Join Stage*, each worker interlinks the assigned pairs of source and target partitions.

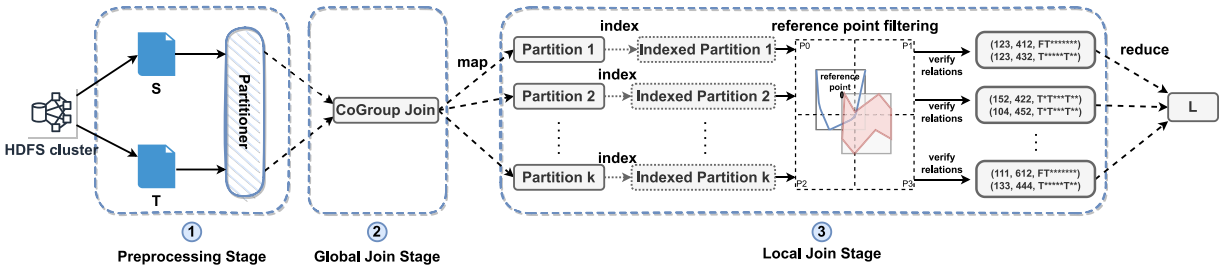

Fig. 4. The three-step pipeline of parallel, batch algorithms in JedAI-spatial.

4.2.1. Batch algorithms

JedAI-spatial conveys the following approaches:

• GeoSpark [30]. This algorithm is now part of *Apache Sedona* [31]. During Preprocessing, it uses sampling to partition the input data with a KDB-Tree or a Quadtree. The geometries that are not covered by the index are added to an overflow partition. The overlapping source and target partitions are assigned to the workers, during the Global Join Stage. The Local Join Stage verifies the candidate pairs through a nested loop join or indexes the source geometries with an R-Tree or a Quadtree that is then probed by the target ones.

• Spatial Spark [32]. It entails two functionalities: (i) The *broadcast join* supports up to 2 GB of source data. During the Preprocessing Stage, the source dataset is indexed by an R-Tree, which is then broadcast as a read-only variable to all workers. Every worker also receives a disjoint partition of the target geometries. The Global Join Stage is skipped. During the Local one, every worker iterates over its target geometries, retrieves the candidate source ones from the R-Tree and verifies those intersecting the target MBR. (ii) The *partition join* overcomes the size limit of the broadcast join by implementing all three steps in Fig. 4. The Preprocessing Stage indexes the entire input data (in case of a Fixed Grid Partition, whose dimensions, $dim_X \times dim_Y$ are defined by the user) or a sample of the source and target data (in case of Binary Split or Sort Tile Partitions, which use an R-Tree). The second stage assigns the overlapping source and target partitions to the same worker so that their candidate pairs are verified locally, during the third stage.

• Magellan [33]. It relies on the Z-Order Curves, which define an Equigrid on the Earth's surface during the Preprocessing Stage. The number of tiles in this grid is determined as 2^p, where p is the precision parameter that is set by the user. The higher this parameter is, the more fine-grained is the resulting Equigrid index. The Global Join Stage sends to the same workers the source and target geometries that intersect the same tiles. During the Local Join Index, every worker checks every candidate pair and verifies those with intersecting MBRs.

• Location Spark [34,35]. Its Preprocessing partitions the source and target datasets using a Grid, R-Tree or Quadtree index. Then, its Query Plan Scheduler performs a skew analysis in order to partition the data as evenly as possible, balancing the workload among the workers. In essence, it repartitions the skewed partitions, which include at least twice as many geometries as the smallest one. After joining the overlapping source and target partitions during the Global Join Stage, a local index is constructed for the source geometries of every worker using an R-Tree, a QuadTee or an EquiGrid. The Local Join Stage probes the index with the target geometries and verifies the candidates with intersecting MBRs.

• Parallel GIA.nt [2]. The Preprocessing estimates the average width and height of the source geometries. These dimensions, which are broadcast to all workers, define the Equigrid that partitions both input datasets. The next stage joins the overlapping source and target partitions, while the Local Join Stage creates an Equigrid of the source geometries inside every worker, using the broadcast dimensions. The target geometries query the index to retrieve the candidates with intersecting MBRs, which are then verified. The reference point technique eliminates all repeated verifications.

4.2.2. Progressive algorithms

JedAI-spatial parallelizes all serial progressive algorithms described in Section 4.1.2. The Preprocessing and Global Join Stage are identical with Parallel GIA.nt. Then, the overall budget BU is split among the partitions assigned to every worker based on the portion of candidate pairs it involves. The Local Join Stage applies the progressive algorithm to the data assigned to every worker, using the corresponding local budget.

5. Remaining architectural components

5.1. Auxiliary components

We now describe the rest of the components in Fig. 3, which play an important role in the system characteristics.

• Data Model. This component implements the classes and the data structures that lie at the core of JedAI-spatial. The cornerstone is the `GeometryProfile` class, which supports all heterogeneous data formats mentioned in Section 3. This is accomplished by representing every geometry as a set of name-value pairs, which capture the textual information about an entity, coupled with a `Geometry` object of the JTS library that is accompanied by its MBR and the method for computing an Intersection Matrix. This simple, yet versatile `GeometryProfile` class also facilitates the visualization and inspection of input data via JS-gui.

• Documentation. This component essentially corresponds to a Java interface that is implemented by all algorithms. The interface conveys methods providing textual information about the most important aspects of each algorithm: its name, a summary of its functionality, the name of every configuration parameter, a short description of every parameter, the domain of every parameter (i.e., its default, minimum and maximum values) as well as the configuration of the current algorithm instantiation. JS-gui provides this information to the user.

• Parameter-configuration. JedAI-spatial facilitates the fine-tuning of any supported algorithm, because a poor parameterization invariably leads to poor performance. Three modes are supported: (i) *Default configuration* a-priori sets all parameters of each algorithm to values that empirically achieve reasonable performance across different datasets. This mode allows lay users to apply the desired pipeline to their data simply by choosing among the available methods. (ii) *Manual configuration* enables power users to fine-tune an algorithm themselves, based on their own experience or on the information provided by the `Documentation` component. (iii) *Grid search* automatically identifies the optimal configuration through a brute-force approach that tries all reasonable values in the domain of each parameter. For batch pipelines, the parameterization minimizing the run-time is selected as the optimal one. For progressive pipelines, the optimal parameters are those maximizing PGR.

• Workflow manager. At the moment, it is responsible for combining the selected filtering technique with the appropriate verification method in terms of execution mode (i.e., serial or parallel). In the future, it will play a crucial role in extending JedAI-spatial with: (i) additional Verification methods like those computing distance relations (e.g., ORCHID [1]), and (ii) multiple filtering methods per pipeline so as to reduce the candidate pairs through the conjunction of their outcomes (this can be carried out efficiently, at no cost in recall).

Table 1
The dataset pairs used in our experiments.

	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	$1.34 \cdot 10^{13}$	$4.49 \cdot 10^{13}$	$8.19 \cdot 10^{13}$	$7.11 \cdot 10^{14}$	$1.38 \cdot 10^{15}$	$8.30 \cdot 10^{15}$
Geometry Pairs with intersecting MBRs	6,310,640	15,729,319	19,595,036	67,336,808	430,597,631	257,075,645
Total Topological Relations	5,635,635	402,936	10,019,188	29,627,279	418,379,333	2,481,027

5.2. Front-end: JS-gui

JedAI-spatial supports users of any experience level, offering two wizard-like user interfaces that simplify its use to a great extent: (i) The command line interface. JS-core produces an executable jar, which when run, guides users in applying the desired pipeline to their data. (ii) The Web application interface. JS-gui is available as a Docker image, which, when deployed, runs a Web application that seamlessly supports serial and parallel execution (based on *Apache Livy* [36]).

In both interfaces, users do not need to write code in order to interlink their spatial data. First, the data reading screen [16,17] asks them to provide the paths of their dataset and the corresponding reading parameters (e.g., the separator character in CSV files). Next, in the algorithm selection screen [16,17], users select the desired pipeline, i.e., serial or parallel, progressive or batch, as well as the desired algorithm among the available ones for the selected pipeline. This applies even to the parallel pipelines that run on Apache Spark. Users can also inspect the input data and store the detected links to a specific path.

JedAI-spatial also acts as a *workbench*, encompassing a results screen [16,17] that facilitates the comparison between the available algorithms. This screen summarizes the performance of the latest runs with respect to the effectiveness measures (i.e., recall, precision, F1 and PGR) as well as the efficiency ones (i.e., the run-time in total and per workflow step). The workbench functionality also allows for examining the impact of configuration parameters on the performance of a particular algorithm (e.g., by changing the granularity of the grid index).

6. Experimental analysis

We now examine the relative performance of serial and parallel batch algorithms as well as of serial progressive algorithms.

Experimental Setup. All experiments were carried out on a server with Intel Xeon E5-4603 v2 @ 2.20 GHz, 32 cores (16 physical), 4 NUMA nodes and 128 GB RAM. The serial methods are implemented in Java 15 and the parallel ones in Scala 2.11.12 and Apache Spark 2.4.4. For each time measurement, we performed 5 repetitions and took the average.

We used 8 real datasets that are popular in the literature [2,28,37,38]. They comprise real data from the US Census Bureau TIGER files and OpenStreeMap (see [17] for details). These datasets are combined into 6 pairs in Table 1.

Serial batch processing. Given that all serial batch algorithms produce the same results (i.e., they detect all topological relations), we exclusively assess their relative time efficiency w.r.t. the *filtering time*, $\mathbf{t_f}$, and the *verification time*, $\mathbf{t_v}$.

First, we perform a weak scalability analysis, examining how the run-time increases with the increase in the size of the input data. We split D_1 into 10 subsets of increasing size, from 10% of source and target geometries to 100% with a step of 10%. The number of related pairs increases in proportion to the dataset size. The resulting t_f and t_v appear in Figs. 5 and 6, respectively. We observe that t_f amounts to few seconds for all algorithms, even when processing the entire D_1. The reason is that Filtering constitutes a quick process that considers exclusively the MBR of the input geometries, thus disregarding their actual complexity. Yet, it reduces the number of candidates by several orders of magnitude, as shown in Table 1.

The algorithms that consider only the source dataset when building their index are much faster than those iterating over both input datasets. The former category includes (Static) GIA.nt, Stripe Sweep and the tree-based algorithms. The static variants of the grid-based algorithms are significantly faster, as they save the cost of deriving the index granularity from the characteristics of the input datasets — they merely index them. Finally, the filtering time of each partition-based algorithm is practically stable, regardless of the underlying data structure (List Sweep or Striped Sweep for Plane Sweep and PBSM, hash map or STR-Tree for Stripe Sweep). Overall, *Quadtree and Stripe Sweep have the fastest Filtering.*

In Fig. 6, t_v is two orders of magnitude larger than t_f in Fig. 5, due to the complexity of the input geometries, which determines the cost of calculating each Intersection Matrix. *(Static) RADON, Plane Sweep and PBSM are faster (in this order)*, because they a-priori load the target geometries into main memory. Plane Sweep and PBSM should be combined with Stripes rather than a Linked List to reduce the maintenance overhead. CR-Tree is excluded, because its t_v over the smallest subset is 235 min, exceeding the time required by most algorithms even for D_1. The reason is the high cost of retrieving the candidates for every target geometry, due to the compression of MBRs. Finally, we should stress that (Static) RADON is faster than (Static) GIA.nt by 5% to 12%, which is in contrast with their relative performance in [2], due to the significant impact of the implementation improvements we have incorporated in JedAI-spatial. *Yet, GIA.nt involves the fastest verification among the algorithms that read the target geometries from the disk, with Stripe Sweep being slightly slower.*

We now compare the same algorithms over D_1 to D_3. Their t_f (in seconds) and t_v (in hours) are reported in Fig. 7, on the left and the right respectively. We exclude D_4 to D_6, because of very high run-times. We also exclude CR-Tree and PBSM/Plane Sweep with a Linked List, due to their poor weak scalability. (Static) RADON, Plane Sweep and PBSM cannot process D_3, because the available 128 GB of RAM do not suffice for loading both S and T in main memory.

The results verify some patterns of the weak scalability analysis. The static grid-based algorithms have a t_v identical with their dynamic counterparts, but a lower t_f, because they do not go through the input geometries when determining the dimensions of their Equigrid. PBSM's Filtering is significantly slower than that of Plane Sweep, because the latter sorts the input geometries just once. Instead, PBSM sorts the input geometries inside every tile. Due to their coarse granularity, its tiles contain a large number of geometries, yielding an overall computational cost that is higher than Plane Sweep. On the other extreme lies Strip Sweep, which has the simplest filtering phase, yielding consistently the lowest filtering time among all methods, followed by Quadtree in close distance. R-Tree involves the next fastest Filtering, as it is outperformed only by Static GIA.nt.

Regarding t_v, (Static) RADON is consistently the fastest algorithm, followed in close distance by Plane Sweep. PBSM is a bit slower, because its coarse-grained tiles involve a large number of redundant candidate pairs that are filtered out by the reference point technique. Among the algorithms that index only the source dataset, Stripe Sweep

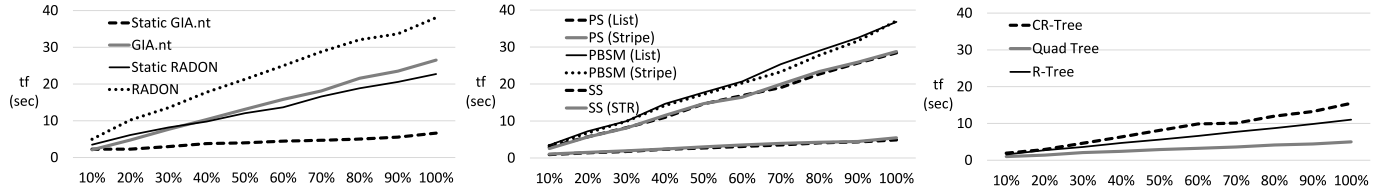

Fig. 5. Weak scalability analysis of the serial batch algorithms w.r.t. Filtering time (s).

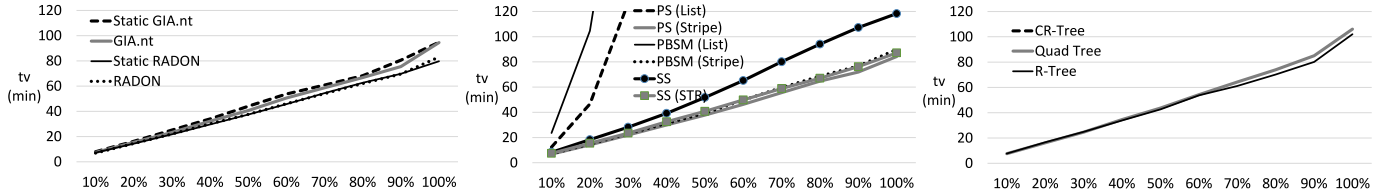

Fig. 6. Weak scalability analysis of the serial batch algorithms w.r.t. Verification time (min).

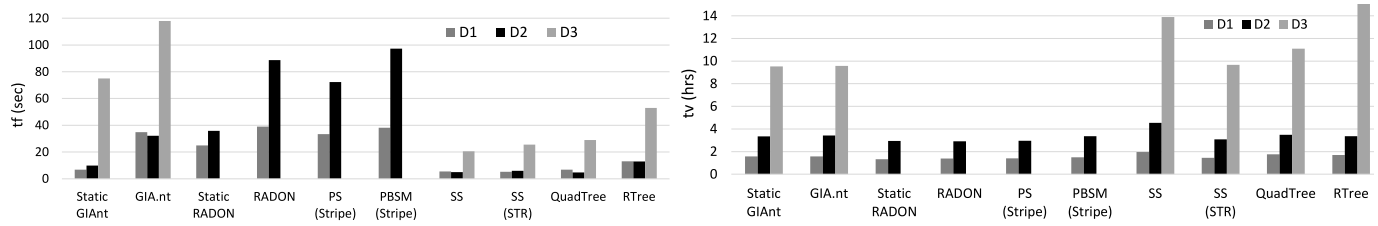

Fig. 7. Filtering (left) and verification (right) time per serial, batch algorithm in D_1-D_3.

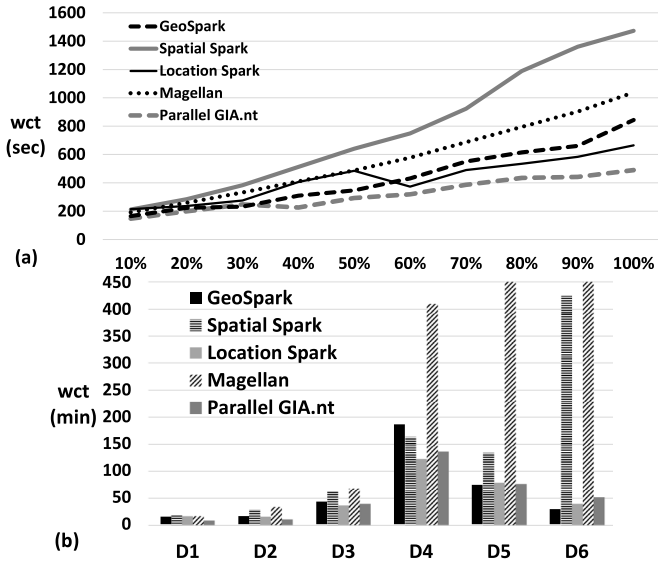

Fig. 8. (a) Weak scalability of parallel batch algorithms, and (b) performance of the parallel batch algorithms over all datasets in Table 1.

with STR is the fastest one, being slower than RADON by 7.1% and 5.5% over D_1 and D_2, respectively. This difference corresponds to the overhead of reading the target dataset from the disk. (Static) GIA.nt is slower than RADON by ~16% in all cases, because it creates a much larger number of fine-grained tiles compared to Stripe Sweep. As a result, (Static) GIA.nt examines the content of many more tiles while processing each $t \in T$.

Finally, Quadtree and RTree perform almost as well as GIA.nt in D_1 and D_2. Over D_3, though, Quadtree is 16% slower than GIA.nt, whereas R-Tree is the slowest algorithm by far, with its t_v (57 h) exceeding the scale of the vertical axis. For both algorithms, this is caused by their sensitivity to their configuration parameters and the large number of overlapping MBRs in D_3. As a result, the range search for candidates per $t \in T$ visits numerous subtrees recursively, yielding a high time complexity that deviates from the average one.

To conclude, the most robust serial, batch algorithms are (Static) GIA.nt and Strip Sweep STR. Their simple, but effective filtering phase scales linearly with the size of the input data, after excluding the effect of a constant overhead in all dataset sizes. Their robust index allows for fast verification, unlike methods like Quadtree and R-Tree, which perform well only after fine-tuning. Finally, RADON, Plane Sweep and PBSM provide competitive run-times for datasets small enough to fit into main memory, given that their Verification phase saves the cost of loading the target dataset from the disk on-the-fly.

Parallel batch processing. To assess the relative run-time of the parallel batch algorithms in Section 4.2.1, we perform a weak scalability analysis using the same subsets of D_1 as in Figs. 5 and 6. After preliminary experiments, we fine-tune the considered algorithms as follows: GeoSpark is coupled with KDB-Tree partitioning and local indexing with R-Tree, Spatial Spark with a 512 × 512 Fixed Grid Partitioning and Location Spark with Quadtree partitioning and local indexing with R-Tree. For Magellan, we set the precision parameter to 20. For Parallel GIA.nt, we changed the source with the target dataset.

Fig. 8(a) reports the corresponding overall wall-clock times (in seconds). *Parallel GIA.nt is consistently the fastest algorithm, with Location Spark following in close distance over the largest subsets*, where its skew

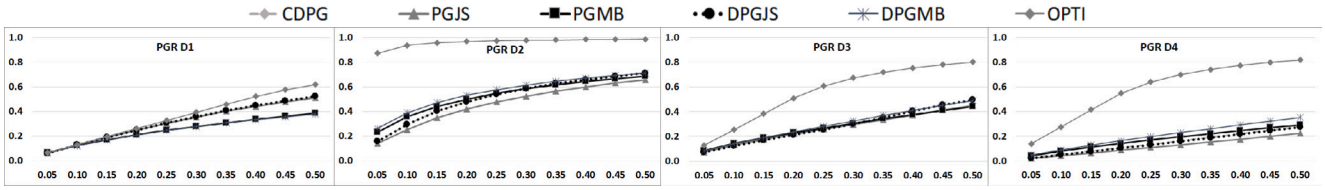

Fig. 9. Performance of the main serial progressive algorithms over D_1-D_4 using as budgets all portions of candidate pairs in $[0.05, 0.50]$ with a step of 0.05.

analysis bears fruit, *leaving GeoSpark in the third place*. These algorithms require less than half the overall run-time of Spatial Spark, with Magellan lying in the middle of these two extremes. All algorithms scale linearly with the input size, after excluding the effect of a constant overhead in all dataset sizes.

All algorithms are much faster than the serial ones, especially over the larger subsets, where Spark's overhead pays off: for the entire D_1, the slowest parallel algorithm (Spatial Spark) is 3.2 times faster than the best serial one (RADON).

Next, we investigate the relative wall clock time of all parallel algorithms over all dataset pairs in Table 1. After preliminary experiments, we applied the same configurations as in the weak scalability analysis to all datasets. The only exceptions are Spatial Spark, which is now combined with 512 × 512 Sort Tile Partitioning, and Parallel GIA.nt, which uses its default configuration, setting the smallest dataset as the source one.

The results appear in Fig. 8(b), exhibiting similar patterns as in the weak scalability analysis. Magellan is consistently the slowest approach, with its run-time increasing disproportionately from D_4 on (for D_6, its execution was actually terminated after 24 h). The reason is that its Preprocessing Stage assigns geometries in significantly more partitions than the rest of the algorithms, thus yielding a very high overhead for the two subsequent phases of the parallel framework in Fig. 4. The second slowest approach is Spatial Spark, because its sampling-based Sort Tile partitioning scales poorly to large datasets, especially D_5 and D_6. The rest of the algorithms yield similar run-times, with each one being the fastest in two datasets: Parallel GIA.nt in D_1-D_2, Location Spark in D_3-D_4 and GeoSpark in D_5-D_6.

Overall, Geospark, Location Spark and Parallel GIA.nt are the most time efficient parallel batch algorithms. The first two depend heavily on their parameter configuration, but Parallel GIA.nt is quite robust with its default configuration, ranking at least as the second best algorithm in all datasets, but D_6.

Serial budget-aware processing. We now compare the best progressive algorithms according to [2] over D_1-D_4. These are: Progressive GIA.nt in combination with the Jaccard (JS) and the $MBRO$ similarity weighting schemes, denoted by PGJS and PGMB, resp., their dynamic counterparts, DPGJS and DPGMB, and Composite Dynamic Progressive GIA.nt, CDPG, which uses JS as the primary scheme and $MBRO$ as the secondary one to break the ties. Note that JS normalizes the number of tiles shared by two geometries by the number of tiles intersecting each geometry, while $MBRO$ returns the normalized overlap of the MBRs of the two geometries. As baseline, we use the optimal progressive algorithm (OPTI), which verifies all topologically related pairs before the non-related ones. For every dataset, we report the performance with respect to PGR for all budgets in the interval $[0.05 \cdot |C|, 0.50 \cdot |C|]$ with a step of 0.05, where $|C|$ denotes the set of candidate pairs. For precision and recall, please refer to [17] and the Appendix in the supplemental material. The results appear in Fig. 9.

In D_1, the best performance is consistently achieved by CDPG, with PGJS and DPGJS following in close distance. These methods rely on JS, outperforming those relying on $MBRO$, i.e., PGMB and DPGMB. The reason is the large proportion of pairs satisfying the relation `touches` (~64.6% [17]), which is hard to be detected by $MBRO$; the bounding rectangles of touching geometries typically have very low overlap and, thus, $MBRO$ assigns extremely low weights to them. Nevertheless, all methods achieve very high performance that is close to the optimal one, due to the relatively large portion of qualifying pairs (~38% of all pairs with intersecting MBRs [17]). In fact, the distance of CDPG from the optimal performance increases with the budget, but amounts to just 8.3%, on average.

In D_2, DPGMB outperforms all other algorithms to a significant extent. PGMB lies in the second place, with its PGR being lower by 5.8%, on average, across all budgets. DPGJS and CDPG exhibit similar behaviors, underperforming DPGMB by 10.7%, on average. PGJS yields the worst performance, which is lower by 20% than DPGMB, on average. In all cases, the smaller the budget is, the higher are differences with CDPG. The distance of DPGMB from OPTI is very high, i.e, ~42.5% on average, due to the heavy class imbalance [17].

Similar to D_2, in D_3, DPGMB and PGMB (in that order) outperform the other algorithms across all budgets. Their average distance from the optimal one is significantly higher than D_1 and D_2, exceeding 40% for all evaluation measures. This should be attributed to the topological relations that dominate the qualifying pairs in D_3, but are underrepresented or absent from D_1 and D_2: `CoveredBy`, `Overlaps`, `Within` and `Equals`.

In D_4, DPGMB and PGMB (in that order) lie between the optimal approach and the progressive methods that leverage JS. In fact, their average distance from the former exceeds 64% for all evaluation measures, while they outperform CDPG, DPGJS and PGJS by 27.7% and 33.8% in terms of precision/recall and PGR, respectively. The high distance from the ideal solution is caused by the same types of topological relations as in D_3. The superiority of $MBRO$ over JS is caused by the polygons that are exclusively contained in D_4.

Overall, we can conclude that *Dynamic Progressive GIA.nt is the best serial progressive algorithm when combined with MBRO weights. The only exception applies to datasets abounding in* `touches` *relations. The more geometries with intersecting MBRs are topologically related, the closer is the performance of Dynamic Progressive GIA.nt to the optimal one.*

7. Related work

In the Semantic Web domain, there are three related systems:

(1) *Silk* [10] constitutes an open-source, generic framework for Link Discovery that comprises a specialized component for Geospatial Interlinking, called Silk-spatial [12]. It exclusively supports a batch, parallel method that runs on Apache Hadoop (http://hadoop.apache.org). Its Filtering relies on a static, coarse-grained Equigrid, whose dimensions are defined by the user. Its Verification computes one topological relation per run.

(2) *LIMES* [11] is an open-source, generic framework for Link Discovery with two algorithms for Geospatial Interlinking: ORCHID [1], which detects proximity relations in an efficient way, and RADON [13], which detects topological relations. RADON's Filtering employs a dynamic Equigrid, whose granularity depends on the input data, while its Verification employs a hash map that maintains all examined pairs in memory to avoid repeated computations. Due to this data structure, RADON has been parallelized as a multi-core, shared-memory process, rather than a shared-nothing, MapReduce approach. Its Verification computes all relations at once [18].

(3) *stLD* [14,15] is a proprietary system for Geospatial Interlinking. It is limited to batch approaches, conveying a variety of algorithms,

such as R-Tree, static Equigrid as well as hierarchical grid. Similar to JedAI-spatial and GIA.nt [2], its algorithms are capable of loading only the source dataset in main memory, while reading the target one on-the-fly. stLD also supports massive parallelization on Apache Flink [39]. Its Verification supports both proximity and topological relations, but computes a single relation per run.

None of these systems supports progressive algorithms, unlike JedAI-spatial, which conveys all the existing progressive methods [2, 28]. JedAI-spatial is also the first system to combine the state-of-the-art main memory spatial join algorithms, which essentially perform Filtering, with the Verification approach that detects topological relations.

Other relevant tools offer parallelization on top of Apache Hadoop or Spark, supporting a variety of spatial queries, such as distance (range) and kNN queries [7]. However, only their spatial join is applicable in the context of Geospatial Interlinking. Each tool essentially offers a single parallel algorithm for this join. The most recent and advanced systems are GeoSpark [30] (a.k.a., Apache Sedona), Spatial Spark [32], Location Spark [34,35] and Magellan. All their algorithms have been integrated into JedAI-spatial.

8. Conclusions

We presented JedAI-spatial, an open-source system that acts as a library of the state-of-the-art algorithms for Geospatial Interlinking. It incorporates optimized implementations and facilitates users by offering two wizard-like interfaces that assume no expert knowledge. Its benchmarking functionality allows for evaluating the relative performance of the available algorithms and for examining the impact of configuration parameters on performance. We elaborated on its architecture, describing the components of its back- and front-end, and performed a thorough experimental analysis, highlighting the relative performance of all batch algorithms and the serial progressive ones.

JedAI-spatial has been developed in the context of the ExtremeEarth EU project [40]. Since then, it is used and extended in other research projects involving geospatial data, namely AI4Copernicus (https://ai4copernicus-project.eu), DeepCube (https://deepcube-h2020.eu) and STELAR (https://stelar-project.eu). It is also used in master theses and student projects in the postgraduate course on Knowledge Technologies at the University of Athens.

We plan to reimplement JedAI-spatial in Python, adding it into the data science ecosystem, and to extend it with multi-core parallelization of all serial algorithms in a systematic way that ensures nearly linear speedup as more CPU cores are available.

CRediT authorship contribution statement

Marios Papamichalopoulos: Software, Validation. **George Papadakis:** Conceptualization, Methodology, Writing – original draft. **George Mandilaras:** Software, Validation. **Maria Siampou:** Software, Validation. **Nikos Mamoulis:** Writing – review & editing. **Manolis Koubarakis:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We have released the data here: https://zenodo.org/record/6384164 The code is available here: https://github.com/AI-team-UoA/JedAI-spatial.

Acknowledgment

This research was partially funded by the Horizon Europe project STELAR (GA No. 101070122).

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.websem.2024.100817.

References

[1] A. Ngomo, ORCHID - reduction-ratio-optimal computation of geo-spatial distances for link discovery, in: ISWC, 2013, pp. 395–410.
[2] G. Papadakis, G.M. Mandilaras, N. Mamoulis, M. Koubarakis, Progressive, holistic geospatial interlinking, in: WWW, 2021, pp. 833–844.
[3] E. Clementini, P. Di Felice, et al., A small set of formal topological relationships suitable for end-user interaction, in: SSD, 1993, pp. 277–295.
[4] E. Clementini, J. Sharma, et al., Modelling topological spatial relations: Strategies for query processing, Comput. Graph. 18 (6) (1994) 815–822.
[5] M.J. Egenhofer, R.D. Franzosa, Point-set topological spatial relations, Int. J. Geogr. Inf. Sci. 5 (2) (1991) 161–174.
[6] E.P.F. Chan, J.N.H. Ng, A general and efficient implementation of geometric operators and predicates, in: SSD, 1997, pp. 69–93.
[7] V. Pandey, A. Kipf, T. Neumann, A. Kemper, How good are modern spatial analytics systems? Proc. VLDB Endow. 11 (11) (2018) 1661–1673.
[8] D. Sidlauskas, C.S. Jensen, Spatial joins in main memory: Implementation matters!, Proc. VLDB Endow. 8 (1) (2014) 97–100.
[9] B. Sowell, M.A.V. Salles, T. Cao, A.J. Demers, J. Gehrke, An experimental analysis of iterated spatial joins in main memory, Proc. VLDB Endow. 6 (14) (2013) 1882–1893.
[10] A. Jentzsch, R. Isele, C. Bizer, Silk - generating RDF links while publishing or consuming linked data, in: ISWC (Posters & Demos), 2010.
[11] A. Ngomo, S. Auer, LIMES - A time-efficient approach for large-scale link discovery on the web of data, in: IJCAI, 2011, pp. 2312–2317.
[12] P. Smeros, M. Koubarakis, Discovering spatial and temporal links among RDF data, in: Workshop on Linked Data on the Web, LDOW, 2016.
[13] M.A. Sherif, K. Dreßler, P. Smeros, A.N. Ngomo, Radon - rapid discovery of topological relations, in: AAAI, 2017, pp. 175–181.
[14] G.M. Santipantakis, et al., Integrating data by discovering topological and proximity relations among spatiotemporal entities, in: Big Data Analytics for Time-Critical Mobility Forecasting, 2020, pp. 155–179.
[15] G.M. Santipantakis, A. Glenis, C. Doulkeridis, A. Vlachou, G.A. Vouros, Stld: towards a spatio-temporal link discovery framework, in: SBD@SIGMOD, 2019, pp. 4:1–4:6.
[16] M. Papamichalopoulos, G. Papadakis, G. Mandilaras, M. Siampou, JedAI-spatial, Zenodo, 2022, http://dx.doi.org/10.5281/zenodo.6520193, https://github.com/gpapadis/JedAI-spatial.
[17] M. Papamichalopoulos, et al., Three-dimensional geospatial interlinking with JedAI-spatial, 2022, CoRR abs/2205.01905.
[18] A.F. Ahmed, M.A. Sherif, A.N. Ngomo, RADON2 - a buffered-intersection matrix computing approach to accelerate link discovery over geo-spatial RDF knowledge bases, in: OM@ISWC, 2018, pp. 197–204.
[19] M. Villamizar, O. Garces, et al., Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS lambda architectures, Serv. Oriented Comput. Appl. 11 (2) (2017) 233–247.
[20] T. Saveta, E. Daskalaki, G. Flouris, I. Fundulaki, M. Herschel, A.N. Ngomo, LANCE: piercing to the heart of instance matching tools, in: ISWC, 2015, pp. 375–391.
[21] T. Brinkhoff, H. Kriegel, B. Seeger, Efficient processing of spatial joins using R-trees, in: SIGMOD, 1993, pp. 237–246.
[22] J.M. Patel, D.J. DeWitt, Partition based spatial-merge join, in: SIGMOD, 1996, pp. 259–270.
[23] J. Dittrich, B. Seeger, Data redundancy and duplicate detection in spatial join processing, in: ICDE, 2000, pp. 535–546.
[24] S.T. Leutenegger, J.M. Edgington, M.A. López, STR: a simple and efficient algorithm for R-tree packing, in: ICDE, 1997, pp. 497–506.
[25] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: SIGMOD, 1984, pp. 47–57.
[26] R.A. Finkel, J.L. Bentley, Quad trees: A data structure for retrieval on composite keys, Acta Inform. 4 (1974) 1–9.
[27] K. Kim, S.K. Cha, K. Kwon, Optimizing multidimensional index trees for main memory access, in: SIGMOD, 2001, pp. 139–150.
[28] G. Papadakis, G.M. Mandilaras, N. Mamoulis, M. Koubarakis, Static and dynamic progressive geospatial interlinking, ACM TSAS 8 (2) (2022).
[29] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, M. Guo, Simba: Efficient in-memory spatial analytics, in: SIGMOD, 2016, pp. 1071–1085.

[30] J. Yu, J. Wu, M. Sarwat, GeoSpark: a cluster computing framework for processing large-scale spatial data, in: SIGSPATIAL, 2015, pp. 1–4.
[31] The Apache Software Foundation, Apache Sedona, 2022, https://sedona.apache.org.
[32] S. You, J. Zhang, L. Gruenwald, Large-scale spatial join query processing in cloud, in: CloudDM Workshop, 2015, pp. 34–41.
[33] R. Sriharsha, Magellan: Geospatial analytics using spark, 2021, https://github.com/harsha2010/magellan.
[34] M. Tang, Y. Yu, Q.M. Malluhi, M. Ouzzani, W.G. Aref, LocationSpark: A distributed in-memory data management system for big spatial data, Proc. VLDB Endow. 9 (13) (2016) 1565–1568.
[35] M. Tang, Y. Yu, A.R. Mahmood, Q.M. Malluhi, M. Ouzzani, W.G. Aref, Location-Spark: In-memory distributed spatial query processing and optimization, Front. Big Data 3 (2020) 30.
[36] The Apache Software Foundation, Apache livy: A REST service for apache spark, 2022, https://livy.apache.org.
[37] A. Eldawy, M.F. Mokbel, SpatialHadoop: A MapReduce framework for spatial data, in: ICDE, 2015, pp. 1352–1363.
[38] D. Tsitsigkos, P. Bouros, N. Mamoulis, M. Terrovitis, Parallel in-memory evaluation of spatial joins, in: SIGSPATIAL, 2019.
[39] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink™: Stream and batch processing in a single engine, IEEE Data Eng. Bull. 38 (4) (2015) 28–38.
[40] D.H. Hagos, T. Kakantousis, V. Vlassov, S. Sheikholeslami, et al., ExtremeEarth meets satellite data from space, IEEE J. Sel. Top. Appl. Earth Obs. Remote. Sens. 14 (2021) 9038–9063.