NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

MSc THESIS

JedAI-spatial: a system for 3-dimensional Geospatial Interlinking

Marios G. Papamichalopoulos

Supervisors **Giorgos Papadakis,** Associate Researcher
Manolis Koubarakis, Professor

ATHENS

OCTOBER 2021

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

JedAI-spatial: ένα σύστημα για τρισδιάστατη Διασύνδεση Γεωχωρικών Δεδομένων

Μάριος Γ. Παπαμιχαλόπουλος

Επιβλέποντες: **Γεώργιος Παπαδάκης,** Συνεργαζόμενος Ερευνητής
Μανόλης Κουμπαράκης, Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2021

MSc THESIS

JedAI-spatial: a system for 3-dimensional Geospatial Interlinking

Marios G. Papamichalopoulos
S.N.: CS3.19.0006

SUPERVISORS: **Giorgos Papadakis,** Associate Researcher
Manolis Koubarakis, Professor

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Manolis Koubarakis,** Professor
Dimitrios Gounopoulos, Professor
Alexandros Ntoulas, Assistant Professor

OCTOBER 2021

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

JedAI-spatial: ένα σύστημα για τρισδιάστατη Διασύνδεση Γεωχωρικών Δεδομένων

Μάριος Γ. Παπαμιχαλόπουλος
Α.Μ.: CS3.19.0006

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γεώργιος Παπαδάκης,** Συνεργαζόμενος Ερευνητής
Μανόλης Κουμπαράκης, Καθηγητής

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Μανόλης Κουμπαράκης,** Καθηγητής
Δημήτριος Γουνόπουλος, Καθηγητής
Αλέξανδρος Ντούλας, Επίκουρος Καθηγητής

ΟΚΤΩΒΡΙΟΣ 2021

ABSTRACT

Geospatial data constitutes a considerable part of Semantic Web data, but so far, its sources lack enough links in the Linked Open Data cloud. Geospatial Interlinking aims to cover this gap by associating geometries with established topological relations, such as those of the Dimensionally Extended 9-Intersection Model. Various algorithms have already been proposed in the literature for this task.

In the context of this master thesis, we develop **JedAI-spatial**, a novel, open-source system that organizes the main existing algorithms according to three dimensions:

i. Space Tiling distinguishes interlinking algorithms into *grid-*, *tree-* and *partition-based*, according to their approach for reducing the search space and, thus, the computational cost of this inherently quadratic task. The former category includes Semantic Web techniques that define a static or dynamic EquiGrid and verify pairs of geometries whose minimum bounding rectangles intersect at least one common cell. Tree-based algorithms encompass established main-memory spatial join techniques from the database community, while the partition-based category includes variations of the cornerstone of computational geometry, i.e., the plane sweep algorithm.

ii. Budget-awareness distinguishes interlinking algorithms into *budget-agnostic* and *budget-aware* ones. The former constitute batch techniques that produce results only after completing their processing over the entire input data, while the latter operate in a pay-as-you-go manner that produces results progressively - their goal is to verify related geometries before the non-related ones.

iii. Execution mode distinguishes interlinking algorithms into *serial ones*, which are carried out using a single CPU-core, and *parallel ones*, which leverage massive parallelization on top of Apache Spark.

Extensive experimental evaluations were performed along these 3 dimensions, with the experimental outcomes providing interesting insights about the relative performance of the considered algorithms.

SUBJECT AREA: Management of Geospatial Data

KEYWORDS: Geospatial Interlinking, Massive Parallelization, Filter-Verification Framework

ΠΕΡΙΛΗΨΗ

Τα γεωχωρικά δεδομένα αποτελούν ένα σημαντικό κομμάτι των δεδομένων του Σημασιολογικού Ιστού (Semantic Web), αλλά μέχρι στιγμής οι πηγές του δεν περιέχουν αρκετούς συνδέσμους στο Linked Open Data cloud. Η Διασύνδεση Γεωχωρικών Δεδομένων (Geospatial Interlinking) έχει ως στόχο να καλύψει αυτό το κενό συνδέοντας τις γεωμετρίες με καθιερωμένες τοπολογικές σχέσεις, όπως αυτές του Dimensionally Extended 9-Intersection Model. Έχουν προταθεί διάφοροι αλγόριθμοι στη βιβλιογραφία για την επίλυση αυτού του προβληματος.

Στο πλαίσιο αυτής της διπλωματικής εργασίας, αναπτύσσουμε το JedAI-spatial, ένα καινοτόμο σύστημα ανοιχτού κώδικα, το οποίο οργανώνει τους κύριους υπάρχοντες αλγορίθμους σύμφωνα με τρεις διαστάσεις:

i. Το Space Tiling διαφοροποιεί τους αλγόριθμους διασύνδεσης σε αυτούς που βασίζονται σε πλέγμα (grid-based), δέντρα (tree-based) ή κατατμήσεις (partition-based), σύμφωνα με την μέθοδο τους για τη μείωση του χώρου αναζήτησης και συνεπώς της τετραγωνικής πολυπλοκότητας αυτού του προβλήματος. Η πρώτη κατηγορία περιέχει τεχνικές Σημασιολογικού Ιστού, η δεύτερη καθιερωμένες τεχνικές για χωρική διασύνδεση (spatial join) στην κύρια μνήμη από την κοινότητα των βάσεων δεδομένων , ενώ η τρίτη περιλαμβάνει παραλλαγές του βασικού αλγορίθμου plane-sweep της υπολογιστικής γεωμετρίας.

ii. Το Budget awareness διαχωρίζει τους αλγόριθμους διασύνδεσης σε budget-agnostic και budget-aware. Οι μέν απαρτίζονται από batch τεχνικές, που παράγουν αποτελέσματα μόνο μετά την επεξεργασία όλων των δεδομένων, ενώ οι δε λειτουργούν με έναν προοδευτικό τρόπο που παράγει αποτελέσματα σταδιακά - ο στόχος τους είναι να επικυρώσουν τις τοπολογικά συσχετιζόμενες γεωμετρίες πριν από τις μη-συσχετιζόμενες.

iii. Η Μέθοδος Εκτέλεσης διαφοροποιεί τους αλγορίθμους σε σειριακούς, οι οποίοι εκτελούνται χρησιμοποιώντας ένα πυρήνα (CPU core), και παράλληλους (parallel), οι οποίοι αξιοποιούν την κατανεμημένη εκτέλεση πάνω στο Apache Spark.

Στα πλαίσια της διπλωματικής πραγματοποιήθηκαν εκτενή πειράματα με τις μεθόδους και των 3 διαστάσεων, με τα πειραματικά αποτελέσματα να παρέχουν μία ενδιαφέρουσα εικόνα όσον αφορά τη σχετική απόδοση των αλγορίθμων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Διαχείριση Γεωχωρικών Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Διασύνδεση Γεωχωρικών Δεδομένων, Παραλληλοποίηση, Πλαίσιο Filter-Verification

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to George Papadakis for helping me throughout the context of this thesis and my professor Manolis Koubarakis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

The basis of this research stemmed from the imperative need of an open-source spatial processing tool for large-scale analysis due to the increased volume of geodata. This tool needs to be friendly for both experienced engineers and ordinary users, as well as robust and optimized for the needs of big data processing. Also, it has to address the different hardware needs of its users by providing both serial and parallel processing. We tackle all these requirements through one tool, JedAI-spatial.

1. INTRODUCTION

The outbreak of Internet of Things (IoT) devices, smartphones, position tracking applications and location-based services have skyrocketed the volume of geospatial data. According to IBM [1], 100TB of weather related data is produced everyday. Uber, an American company providing transportation services, on May 20, 2017 hit the milestone of 5 billion rides among 76 countries [2]. Platforms such as OpenStreetMap[1] provide an open and editable map of the whole world. For these reasons, geospatial data constitute a considerable part of Semantic Web data, but so far, its sources lack enough links in the Linked Open Data cloud [3].

The data science community is stranded in a sea of geospatial data waiting to be analyzed and processed. A robust and large-scale spatial analysis has been imperative more than ever. *Geospatial Interlinking* aims to cover this gap by associating pairs of geometries with established topological relations, such as those of the Dimensionally Extended 9-Intersection Model (DE-9IM) [4]. See Figure 1 for an example.

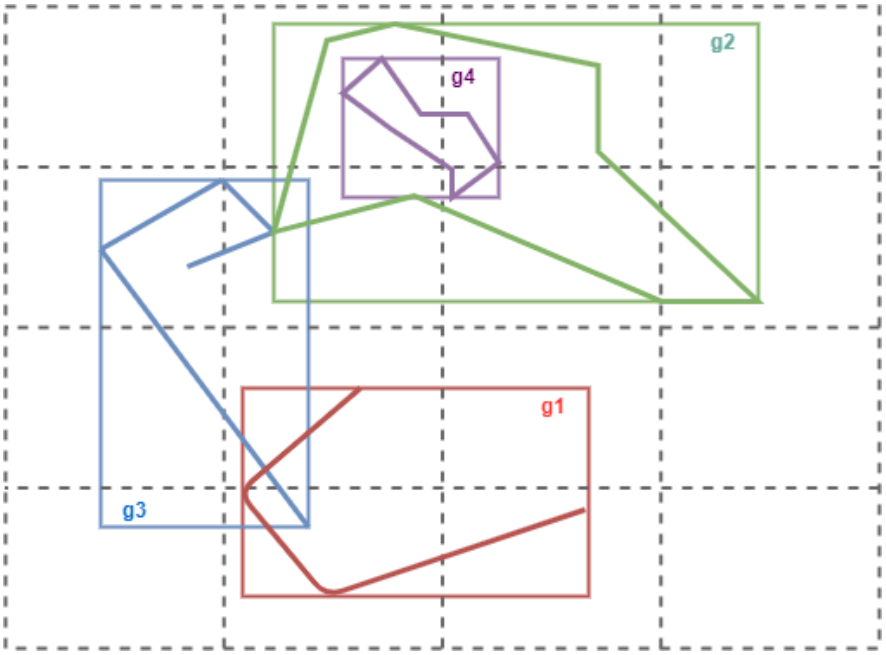

Figure 1. Geospatial Interlinking between LineString g1 which intersects Linestring g3 and touches Polygon g2 which contains Polygon g4.

Two are the main challenges of this task:

1. its inherently quadratic time complexity, because every pair of geometries has to be examined, i.e., $O(n^2)$, where n is the number of input geometries. This is known as a *nested loop join*, a brute-force approach, where two given datasets are joined using two nested loops.
2. the time complexity of examining a single pair of geometries is also high, $O(N \cdot logN)$, where *N* is the size of the union set of their boundary points [5].

To tame the overall high computational cost, various methods based on the Filter-Verification have been proposed in the literature [17], [18], [19]; they reduce the time complexity in favour of space complexity by restricting the search space to pairs of geometries that are likely to be topologically related according to a geospatial index. This is known as a *nested loop index join*.

[1] https://www.openstreetmap.org

The continuous advancement in technology and, hence, price dropping in Random-Access Memory (RAM) modules, gave rise to a plethora of frameworks and algorithms able to run in main memory. Moreover, on-demand cloud services like Amazon Web Services (AWS)[2] or Microsoft's Azure[3] provide reliable hardware for rent and deployment, which can run as a cluster for spatial processing. Software engineers leverage this new standard by constructing algorithms able to run on parallel or distributed environments. For example, Apache Hadoop[4] is a distributed, fault-tolerant, map-reduce framework implemented in Java[5], for large scale processing of big data. It is now considered deprecated, due to its disk-oriented processing, meaning that it continuously reads and writes from disk, involving a high I/O cost. However, its ecosystem is vast and is still used by many applications, like for example the Hadoop Distributed File System (HDFS)[6]. Its successor, Apache Spark[7] [8], improves all of Apache Hadoop's flaws introducing the Resilient Distributed Dataset structure (RDD) and operating both in main memory and disk for faster execution.

Developers, however, are faced with the task of choosing the appropriate execution mode (serial or parallel) and algorithms for geospatial interlinking, based on their infrastructure, data size and data types. For example, some algorithms may be more optimal for polylines than polygons. For some other cases where the volume of spatial data is not that large, a serial experiment would outweigh the effort to set up and tune an Apache Spark cluster, a task which can be time consuming. Also, Spark's leader election for every job may impose a noteworthy overhead for smaller datasets according to [20]. To assist developers and practitioners in the processing of geospatial data, we conduct thorough evaluation experiments that show the limitations and fortes of each method.

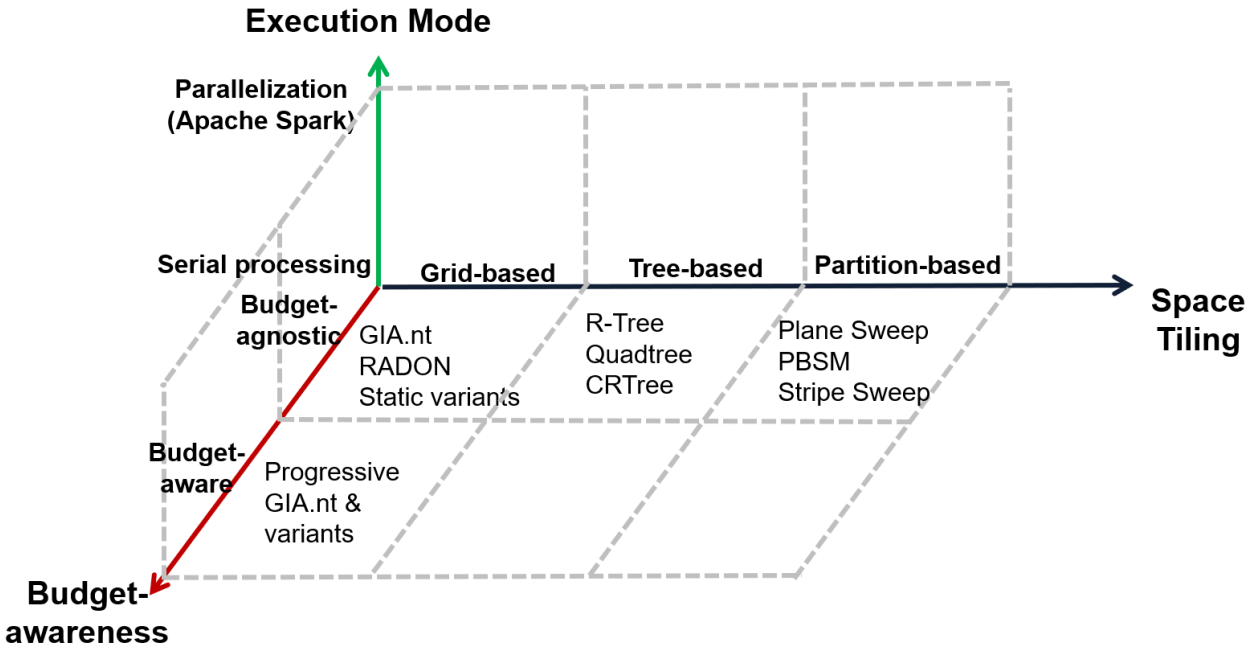

Figure 2. The solution space of Geospatial Interlinking algo-rithms that can be constructed by JedAI-spatial.

[2] https://aws.amazon.com/
[3] https://azure.microsoft.com/en-us/
[4] https://hadoop.apache.org/
[5] https://www.java.com/en/
[6] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[7] https://spark.apache.org/

In the context of this master thesis, we develop **JedAI-spatial**, a novel, open-source system that organizes existing spatial join algorithms according to three dimensions, as shown in Figure 2:

1) **Space Tiling** distinguishes interlinking algorithms into *grid-*, *tree-* and *partition-based*, according to their approach for reducing the search space and, thus, the computational cost of this inherently quadratic task. The former category includes Semantic Web techniques that define a static or dynamic EquiGrid and verify pairs of geometries whose minimum bounding rectangles intersect at least one common cell. The tree-based category encompasses established main-memory spatial join techniques from the database community, while the partition-based category includes variations of the plane sweep algorithm, a cornerstone of computational geometry.
2) **Budget-awareness** distinguishes interlinking algorithms into *budget-agnostic* and *budget-aware* ones. The former constitute batch techniques that produce results only after completing their processing, while the latter operate in a pay-as-you-go manner that produces results progressively - their goal is to verify the topologically related geometries before the non-related ones.
3) **Execution mode** distinguishes interlinking algorithms into *serial ones*, which are carried out on a single CPU-core, and *parallel ones*, which leverage massive parallelization on top of Apache Spark.

Overall, the contributions of this thesis are the following:

- We describe the three-dimensional categorization of JedAI-spatial, explaining the role of every sub-category so as to facilitate the use of its methods as well as its extension with more methods in the future.
- We outline the functionality of every supported method, highlighting our improvements that lead to significantly higher time efficiency.
- We perform a qualitative analysis of the supported methods, identifying the most suitable use cases per method.
- We perform a quantitative analysis of the supported methods through extensive experiments over large, real-world datasets. The empirical results provide useful insights into the pros and cons of every method.
- We have publicly released the code of our system[8] (in Java and Scala[9]) so that it acts as a library for the state-of-the-art algorithms for Geospatial Interlinking. Our implementation conveys improvements for each method we support, which we will discuss in more detail in the following chapters.

The rest of the thesis is structured as follows:

- *Chapter 2 (Background Knowledge)* presents the preliminaries for the thorough understanding of the thesis.
- *Chapter 3 (Serial Implementations)* delves into the famous algorithms for spatial joins that run on a single CPU core.
- *Chapter 4 (Parallel Implementations)* describes the Apache Spark-based algorithms that run in parallel on a cluster.
- *Chapter 5 (Experimental Results)* discusses the experimental results of the previous two sections applied on six massive real world datasets.
- *Chapter 6 (Conclusions and Future Work)* concludes the research conducted and provides some useful future additions to the system.

[8] https://github.com/GeoLinker/GeoLinker
[9] https://www.scala-lang.org/

2. BACKGROUND KNOWLEDGE

JedAI-spatial supports geometries that consist of interior, boundary and exterior (i.e., all points that are not part of the interior or the boundary). These are distinguished into two main types [18]:

- *LineStrings* which constitute one-dimensional geometries formed by a sequence of points and the line segments that connect consecutive points (e.g., geometries g1 and g3 in Figure 1), and
- *Polygons*, which in the simple case are two-dimensional geometries formed by a sequence of points where the first one coincides with the last one (e.g., geometries g2 and g4 in Figure 1).

The implementations in both serial and parallel processing follow three common steps:

1. **Preprocessing Phase**, where the source and target data are read and prepared for the subsequent phases.
2. **Filtering Phase**, which filters out the source and target geometries whose MBRs do not intersect. The rest of the geometry pairs are called *candidate pairs*.
3. **Verification Phase**, which computes the topological relations of the candidate pairs.

JedAI-spatial supports spatial join operations conforming to the *Dimensionally Extended 9-Intersection Model (DE-9IM)* [4] or Clementini Matrix, which shows the spatial relations between two 2D-objects based on their *interior (I)*, *boundary (B)* and *exterior (E)*. DE-9IM is an extension of the Nine-Intersection Model (9IM) or Egenhofer-Matrix considering the dimension type of the intersection. 9IM enhanced the Four-Intersection Model (4IM), which only considered boundary and exterior, by adding the interior. DE-9IM can be used to answer queries such as “Find the countries that touch Greece” or “Which cities does the Attica region contain?”.

A topological relation is described using a *3x3* intersection matrix between the interiors, exteriors and boundaries of the two geometries, where dim denotes the dimension and ∩ the intersection:

$$\text{DE-9IM}(A,\, B) = \begin{bmatrix} \dim\left(I(A) \cap I(B)\right) & \dim\left(I(A) \cap B(B)\right) & \dim\left(I(A) \cap E(B)\right) \\ \dim\left(B(A) \cap I(B)\right) & \dim\left(B(A) \cap B(B)\right) & \dim\left(B(A) \cap E(B)\right) \\ \dim\left(E(A) \cap I(B)\right) & \dim\left(E(A) \cap B(B)\right) & \dim\left(E(A) \cap E(B)\right) \end{bmatrix}$$

The DE-9IM has been standardized by the Open Geospatial Consortium (OGC). It provides ten topological predicates:

1. $Equals(A,\, B)$: the geometries’ interiors are identical and the boundary and exterior of A intersect neither with the boundary nor with the exterior of B.
2. $Disjoint(A,\, B)$: the geometries have no point in common, meaning the interior and boundary of A intersect neither with the interior nor with the boundary of B. The opposite of $disjoint$ is the $intersects$ predicate.
3. $Intersects(A,\, B)$: the geometries have at least a point in common, thus their interiors or boundaries are not disjoint.
4. $Touches(A,\, B)$: the geometries’ boundaries intersect but their interiors do not.
5. $Crosses(A,\, B)$: geometries A and B have some interior points in common but not all, while $dim(A) < dim(B)$ or $dim(B) < dim(A)$, where $dim(g)$ amounts to 0, 1 or 2 if geometry g is a point, a line segment or an area, respectively.
6. $Overlaps(A,\, B)$: geometries A and B have some points in common but not all, while $dim(A) = dim(B)$.

7. $Within(A,\ B)$: geometry A exists inside the interior of geometry B and no points of A lie in the exterior of B.
8. $Contains(A,\ B)$: geometry B is within A.
9. $Covers(A,\ B)$: every point in geometry B lies in the interior or boundary of A and no point of B lies in the exterior of A.
10. $CoveredBy(A,\ B)$: geometry B covers geometry A.

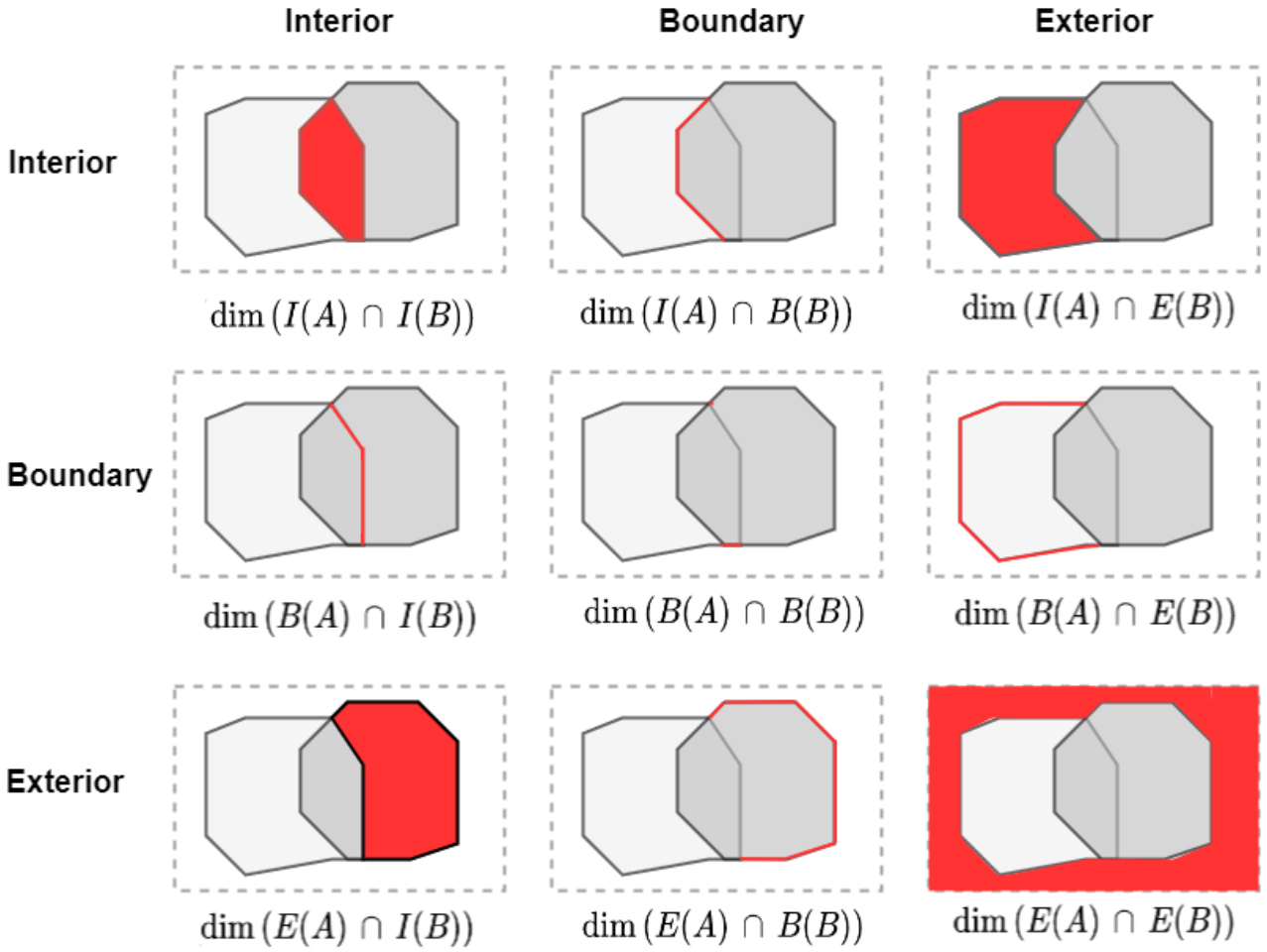

Figure 3. Dimensionally Extended 9-Intersection Model Topological Relations.

The intersection matrix of the preceding predicates is depicted in Figure 3 and Figure 4. Each predicate is a boolean representation of the intersection matrix, where each cell may be $True\ (T)$, $False\ (F)$ and any (*). F denotes the empty set and T corresponds to the dimension of the intersection (i.e., $T \leftrightarrow dim(A,\ B) \in \{0,\ 1,\ 2\}$, as explained above). DE-9IM is used in state-of-the-art spatial databases such as PostGIS[10], Oracle Spatial[11] and in popular libraries like the Java Topology Suite (JTS)[12].

[10] https://postgis.net/
[11] https://www.oracle.com/database/spatial/
[12] https://github.com/locationtech/jts

$$Equals(A,B) = \begin{bmatrix} T & * & F \\ * & * & F \\ F & F & * \end{bmatrix}$$

$$Disjoint(A,B) = \begin{bmatrix} F & F & * \\ F & F & * \\ * & * & * \end{bmatrix}$$

$$Intersects(A,B) = \begin{bmatrix} T & * & * \\ * & * & * \\ * & * & * \end{bmatrix} or \begin{bmatrix} * & T & * \\ * & * & * \\ * & * & * \end{bmatrix} or \begin{bmatrix} * & * & * \\ T & * & * \\ * & * & * \end{bmatrix} or \begin{bmatrix} * & * & * \\ * & T & * \\ * & * & * \end{bmatrix}$$

$$Touches(A,B) = \begin{bmatrix} F & T & * \\ * & * & * \\ * & * & * \end{bmatrix} or \begin{bmatrix} F & * & * \\ * & T & * \\ * & * & * \end{bmatrix} or \begin{bmatrix} F & * & * \\ T & * & * \\ * & * & * \end{bmatrix}$$

$$Crosses(A,B) = \begin{bmatrix} T & * & T \\ * & * & * \\ * & * & * \end{bmatrix} or \begin{bmatrix} 0 & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

$$Overlaps(A,B) = \begin{bmatrix} T & * & T \\ * & * & * \\ T & * & * \end{bmatrix} or \begin{bmatrix} 1 & * & T \\ * & * & * \\ T & * & * \end{bmatrix}$$

$$Within(A,B) = \begin{bmatrix} T & * & F \\ * & * & F \\ * & * & * \end{bmatrix}$$

$$Contains(A,B) = \begin{bmatrix} T & * & * \\ * & * & * \\ F & F & * \end{bmatrix}$$

$$Covers(A,B) = \begin{bmatrix} T & * & * \\ * & * & * \\ F & F & * \end{bmatrix} or \begin{bmatrix} * & T & * \\ * & * & * \\ F & F & * \end{bmatrix} or \begin{bmatrix} * & * & * \\ T & * & * \\ F & F & * \end{bmatrix} or \begin{bmatrix} * & * & * \\ * & T & * \\ F & F & * \end{bmatrix}$$

$$CoveredBy(A,B) = \begin{bmatrix} T & * & F \\ * & * & F \\ * & * & * \end{bmatrix} or \begin{bmatrix} * & T & F \\ * & * & F \\ * & * & * \end{bmatrix} or \begin{bmatrix} * & * & F \\ T & * & F \\ * & * & * \end{bmatrix} \begin{bmatrix} * & * & F \\ * & T & F \\ * & * & * \end{bmatrix}$$

Figure 4. Dimensionally Extended 9-Intersection Model Spatial Predicates.

Note, though, that JedAI-spatial disregards the relation $disjoint$, because it provides no positive information for the relative location of two geometries and because it is not practical to compute it in the case of large input data. The reason is that the vast majority of pairs pertain to this relation, which thus scales quadratically with the input size. Yet, JedAI-spatial is based on a closed-world assumption: the lack of the relation $intersects$ between two geometries implies that they satisfy the relation $disjoint$.

In this context, Geospatial Interlinking can be formally defined as follows [18]:

Given a source dataset S, a target one T, and the set of DE9IM topological relations R (excluding $disjoint$), derive the set of links $L_R = \{(s,r,t) \subseteq S \times T \times R: r(s,t)\}$ from the Intersection Matrix of all geometry pairs.

In the preprocessing process, JedAI-spatial expects the format of its geometries in Well Known Text (WKT), another OGC standard, which represents spatial data in a text form. In Figure 5, there are some examples of geometries encoded in WKT.

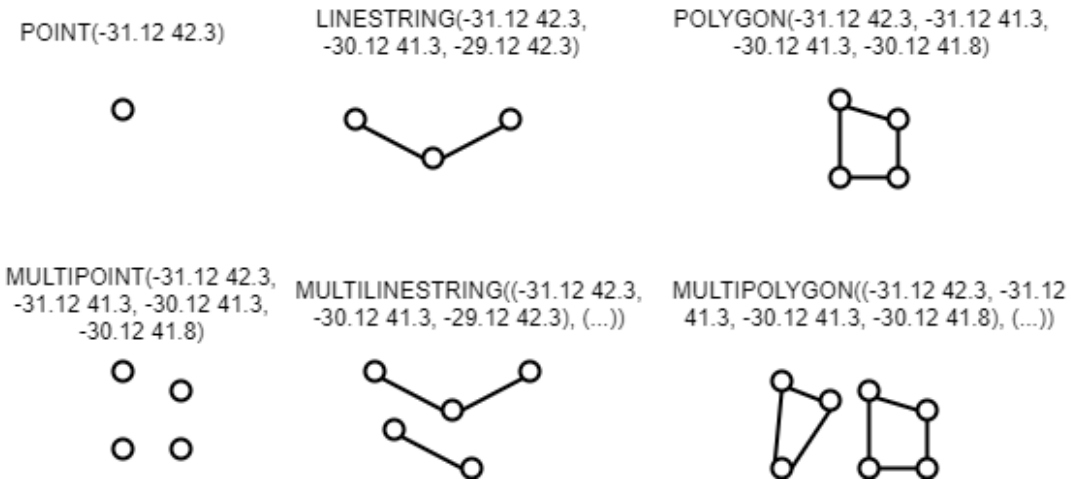

Figure 5. Well Known Text examples.

JedAI-spatial supports a wide variety of formats, both structured and semi-structured, that store geometries in WKT: Comma-Separated Values (CSV), Tab-Separated Values (TSV), GeoJSON, Resource Description Framework (RDF) and JsonRDF file formats.

Apparently, finding the spatial predicates between geometries can be a tough task, especially between MULTIPOLYGONS and MULTILINESTRINGS. A small, but complex

island or lake may need hundreds or thousands of vertices, let alone a country like Greece which consists of hundreds of islands and has a vast coastline. In order to simplify the filtering process, all the geometries are approximated and bounded by the smallest rectangle which completely contains them, called *Minimum Bounding Rectangle* (**MBR**) [28]. MBRs are a classic technique used in R-Trees and other spatial indexes. JTS library has integrated MBR inside the Geometry Object, which lies at the core of JedAI-spatial.

JedAI-spatial also employs a state-of-the-art technique for faster filtering, the *Reference Point Method* (**RPM**) [9]. In the preprocessing stage, each geometry may end up in more than one tile or data partition. As a result, the same pair of geometries might be encountered multiple times. To avoid repeating their verification, in the filtering phase, a pair of geometries is added to the set of candidate pairs if their reference point rf lies within the tile/partition that is currently processed. More formally, the reference point rf between two geometries, s and t, is defined as:

$$rf = (max(s.x_{min}, t.x_{min}), min(s.y_{max}, t.y_{max}))$$

Example 1: The following two geometries, a linestring (blue) and a polygon (red), have been placed in four tiles or partitions that intersect their MBRs. Bear in mind that each tile/partition typically constitutes an independent processing unit. To avoid verifying their relations more than once, each tile/partition calculates the reference point of the geometries and adds them to the set of candidate pairs if and only if the reference point lies within the extent of the tile/partition that is currently processed. In Figure 6, the reference point is located inside tile/partition $P0$ and, thus, only this partition verifies the pair, whereas the rest do not.

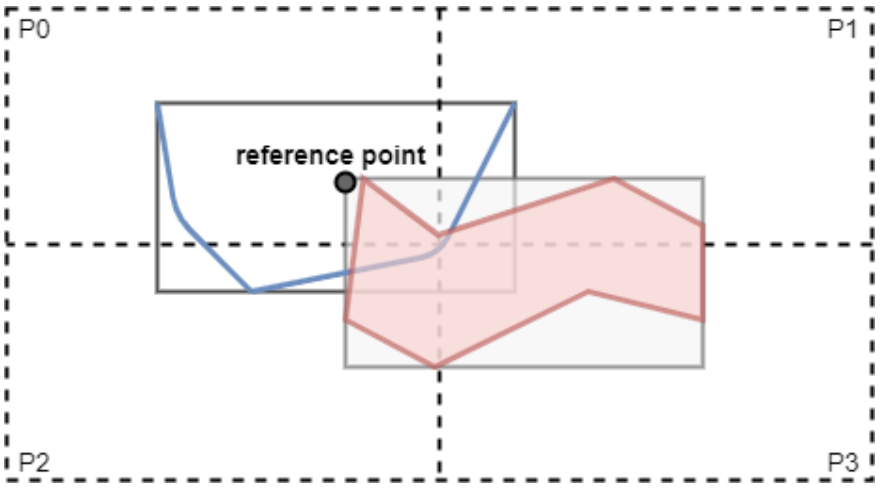

Figure 6. Reference Point Method.

3. SERIAL IMPLEMENTATIONS

This section comprises established sequential, spatial join algorithms which run in main memory based on the analysis by Sowell et al. [6]. In this work, each experiment consists of steps called *ticks*. Ticks are primarily used for simulations on moving data that have two phases: (i) a query phase, which performs spatial queries, and (ii) an update phase, which conducts updates in the data, like a change in velocity, in order to continue to the next tick.

The spatial join techniques are partitioned into four categories depending on the indexing approach (*Static* or *Moving* points) and on the join approach (*Index Nested Loops* or *Specialized*):

- **Static Index**: an index is built over static points at the beginning of the experiment and is not destroyed or updated by new geometries in other ticks.
- **Moving Index**: an index which accommodates moving points preserving their velocity and is updated when a point's velocity changes.
- **Nested Loop Join**: a spatial join approach which queries each target geometry on the spatial index.
- **Specialized**: the techniques that cannot be classified as nested loop join techniques, since they use a more sophisticated spatial join method.

JedAI-Spatial pertains to *static* data and employs most of the algorithms implemented in [6], namely: Plane Sweep [10], Partitioned Based Spatial-Merge Join (PBSM) [11], R-Tree [12], CR-Tree [13].

Additional established algorithms have been implemented: Strip Sweep and Strip STR Sweep, Quad Tree [14], based on the implementation by the JTS library, as well as RADON [17] and GIA.nt [18] along with their static variants.

Following is a table organizing the algorithms implemented by JedAI-spatial in two categories, according to their join approach:

Table 1. Serial Implementations categorized by their spatial join approach.

Nested Loop Join	Specialized
R-Tree, CR-Tree, Quad Tree, RADON, Static RADON, GIA.nt, Static GIA.nt	Plane Sweep, Strip Sweep, Strip STR Sweep, PBSM

All the algorithms have been implemented in Java, which allows JedAI-spatial to be integrated into a web application or a Maven[13]/Gradle[14] dependency so that novice practitioners can run experiments without any programming background and software engineers can use it as a dependency in their own projects.

In the following, we delve into the Filtering step of the serialized algorithms, as they all share the same Preprocessing and Verification step. The latter is based on JTS's method `relate`, which receives as input a pair of geometries and returns as output their Intersection Matrix through an optimized implementation.

3.1. Plane Sweep

The *Plane sweep(sweepline)* algorithms [10] have been prevalent regarding spatial joins. In general, such algorithms sort the geometries in ascending order of their lower boundary on the x or y axis and then move a *vertical or horizontal* sweepline across the space.

[13] https://maven.apache.org/
[14] https://gradle.org/

Plane-sweeping variants utilize a dynamic data structure, called *sweep structure*, in order to save the *active geometries,* whose MBR intersects the sweepline in its current position. As a rule, the memory overhead of such methods includes only the space of the sweep structure, which never overcomes the size $O(\sqrt{N})$, an observation known as the *square-root rule* in the literature [15]. In more detail, given a dataset containing N real-life rectangles, the ones intersected by the sweepline at any given moment, are at most $O(\sqrt{N})$.

Plane sweeping algorithms are usually treated as building blocks for more sophisticated methods like the *Scalable Sweeping-Based Spatial Join (SSSJ)* [10] and *Partition Based Spatial-Merge (PBSM)* [11]. The overall performance of these methods is greatly influenced by the corresponding plane sweeping algorithm that acts as an internal procedure.

Arge et al. [10] present variants of the Generic Plane Sweep and Forward Plane Sweep algorithms depending on their sweep structures. JedAI-Spatial employs two of those variants, the *Algorithm List_Sweep* and *Algorithm Striped_Sweep*.

3.1.1. Algorithm List_Sweep

Algorithm List_Sweep is initialized using two linked-list sweep structures, one for each input dataset, in order to save the active geometries in main memory. Both datasets are sorted on their lower boundary of the x-axis using quick-sort. The following methods are used:

- **`Insert`:** inserts a geometry into the linked list**.**
- **`Delete`:** deletes the expired geometries from the list. Expired are the geometries whose x_{max} is less than the x_{min} given as a parameter to the function.
- **`Query`:** finds the intersection between the geometry given as a parameter and the geometries of the list.
- **`Remove`:** removes the geometry completely from the starting set so that the loop can continue. In the actual code, the geometry counter is increased by one, because the geometries are saved in an array data structure.

Algorithm 1: Generic Plane Sweep - List Sweep

Input: source dataset S, target dataset T
Output: set of links L with the detected topological relations
Sort S and T on lower boundary of x-axis
Let two empty Linked Lists L_s and L_t
while *S and T are not empty* **do**
  Let $s = Head(S)$ and $t = Head(T)$
  if $s.x_{min} < t.x_{min}$ **then**
    $Insert(L_s, s)$
    $Delete(L_t, s.x_{min})$
    $Query(L_t, s)$
    $Remove(S, s)$
  else
    $Insert(L_t, t)$
    $Delete(L_s, t.x_{min})$
    $Query(L_s, t)$
    $Remove(T, t)$
  end
end

Figure 7. Algorithm List_Sweep.

The general idea is to find the intersections between the MBRs of the source and target geometries that are intersected by the same sweepline. To be more precise, to find the intersections between rectangles whose x_{max} is ahead of the sweepline.

Example 2. The algorithm provided in Figure 7 is illustrated in Figure 8. The list for the source and target geometries are Ls and Lt, respectively. Initially, the sweepline encounters the MBR of geometry a, appends a to list Ls and continues with the next iteration since Lt is empty. Then, the sweepline comes across c's MBR. It appends c to Lt and checks if its MBR intersects with that of a. It does not, so it proceeds with the next iteration. In the third iteration, the sweepline encounters geometry b. It appends it to list Ls and removes the expired geometries from list Lt. Since $c.x_{max} < b.x_{min}$, c is removed from the list, which remains empty. At the fourth and last iteration, the sweepline reaches the target geometry d, which is inserted into list Lt. All expired geometries are removed from Ls, hence a is removed. The one left is the source geometry b, which is checked if it intersects with c. Since their MBRs intersect, their DE-9IM relations are verified by computing their Intersection Matrix.

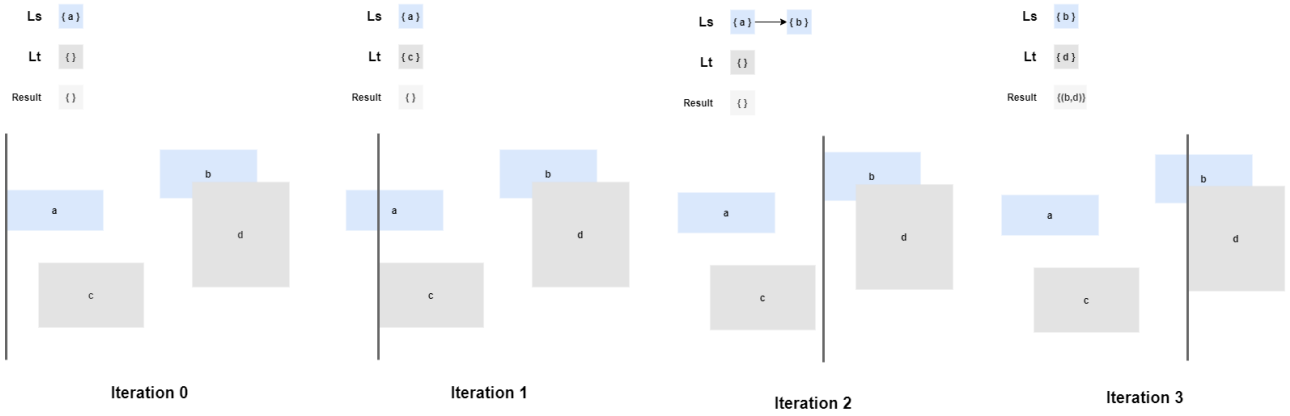

Figure 8. Algorithm List_Sweep Visually.

3.1.2. Algorithm Striped_Sweep

This algorithm is an extension of the Algorithm List_Sweep. Rather than having one List structure, the datasets are divided into n equal-width strips, where each strip uses one List-Sweep Structure.

The length of each strip is key to the performance of the technique. In [10], the length of the strips was calculated experimentally. However, in JedAI-spatial, each strip's width is equal to the average width of the source geometries ($thetaX$). This does not require experimenting in order to find the optimal width, which changes from dataset to dataset, and dwindles worst case scenarios, such as when a geometry expands to a large number of strips imposing a lot of overhead in the verification process. Our approach provides a much safer and robust configuration, as verified by preliminary experiments.

Algorithm 2: Generic Plane Sweep - Striped Sweep

Input: source dataset S, target dataset T
Output: set of links L with the detected topological relations
Sort S and T on lower boundary of x-axis
Calculate the average length of the source geometries $thetaX$
$minX \leftarrow min_{x-axis}(S)$
$maxX \leftarrow max_{x-axis}(S)$
$stripSize \leftarrow widthOfAllGeometries/thetaX$
$numOfStrips \leftarrow (maxX - minX)/stripSize$
$S_s \leftarrow List\_Sweep[numOfStrips]$
$S_t \leftarrow List\_Sweep[numOfStrips]$
while S *and* T *are not empty* **do**
  Let $s = Head(S)$ and $t = Head(T)$
  if $s.x_{min} < t.x_{min}$ **then**
    $Insert(S_s)$
    $Delete(S_t, s.x_{min})$
    $Query(S_t, s)$
    $Remove(S, s)$
  else
    $Insert(S_t, t)$
    $Delete(S_s, t.x_{min})$
    $Query(S_s, t)$
    $Remove(T, t)$
  end
end

Figure 9. Algorithm Striped_Sweep.

In the algorithm above, the `Insert`, `Delete` and `Query` methods are modified to call the underlying List Sweep methods on the strips that the geometry is probed. For example, if there are 16 strips and the geometry spans from the 1st strip to the 3rd one, then an insert would be invoked on the List Sweep structures 0-2.

In the code repository, for both variants of the Plane Sweep algorithm, only one List Sweep and Strip Sweep structure is used for better memory performance.

3.2. Partitioned Based Spatial-Merge Join (PBSM)

This algorithm [11] 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 horizontal line l, computing the Intersection Matrix for each pair of geometries that simultaneously intersect l and overlap on the y-axis. To avoid repeated verifications of the same geometry pairs across different partitions, it uses the reference point technique, verifying every candidate pair only in the partition that contains the top left corner of their intersection (see Figure 6).

Similar to Plane Sweep, JedAI-spatial combines PBSM with a List Sweep or with a Striped Sweep data structure.

3.3. Strip Sweep

This algorithm is a custom implementation inspired by the *Algorithm Striped_Sweep*. It calculates the number of strips in an identical way (i.e., average width of the source geometries), but avoids sorting both datasets on their lower boundary of x-axis. The reason is that Strip Sweep indexes only the source dataset, dividing it into strips, and then probes each target geometry on the corresponding strips. The target geometries are read from the disk, one by one, thus minimizing the memory requirements. As a result, Strip Sweep is able to process much larger datasets than Plane Sweep using the same infrastructure.

In more detail, the Strip Sweep algorithm works as follows:

Algorithm 3: Strip Sweep

Input: source dataset S, target dataset T
Output: set of links L with the detected topological relations
Divide the source dataset into equal width strips ($thetaX$)
while T *is not empty* **do**
  Let $t = Head(T)$
  Find the strips t resides to
  Let the source geometries within the strips S'
  while S' *is not empty* **do**
    Let $s' = Head(S')$
    if $s'.intersects(t)$ **then**
      $L+ = verifyRelations(s', t)$
    end
  end
end

Figure 10. Strip Sweep Algorithm.

Example 3. The functionality of the above algorithm is depicted in the following figures. Firstly, the source dataset is indexed in equal width strips as seen in Figure 11. Then each geometry in the target dataset is read and mapped to the appropriate strips, as shown in Figure 12. All geometries within those strips are considered as *candidates* and are verified if their MBRs intersect, in Figure 13 and in Figure 14, respectively.

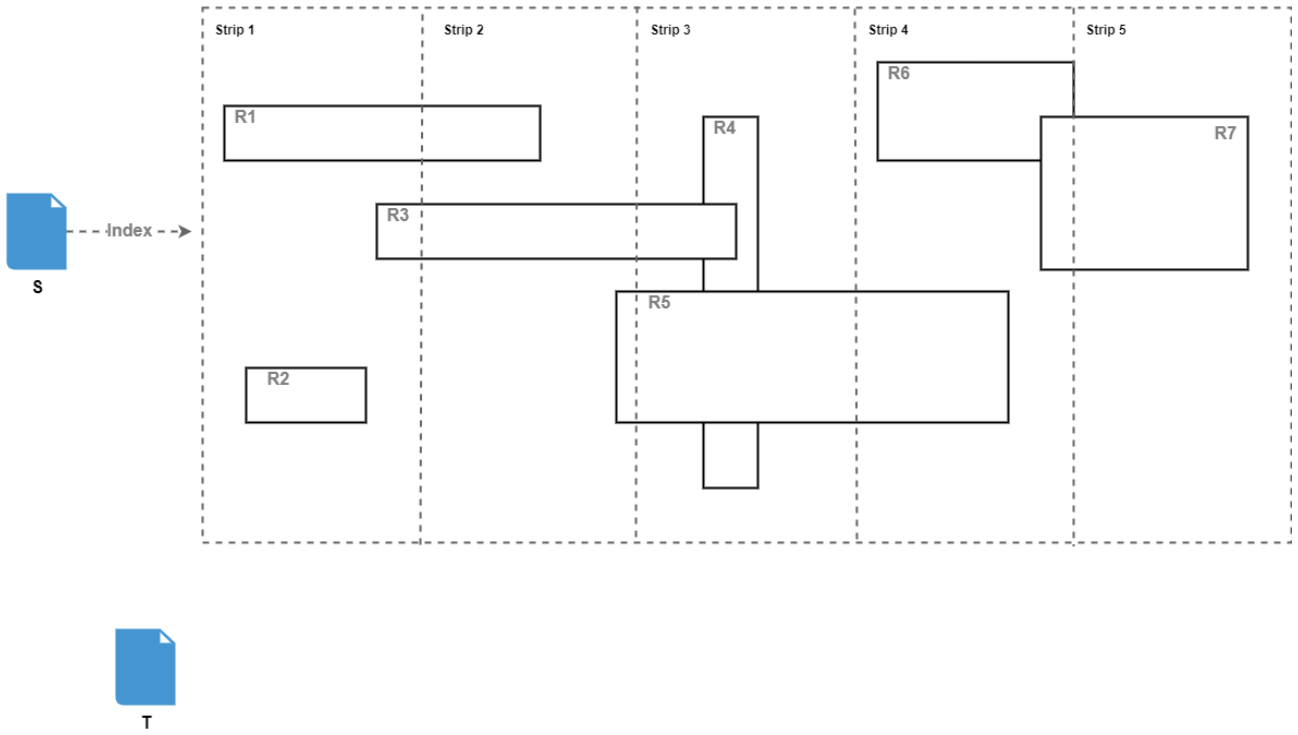

Figure 11. Strip Sweep: Partitioning the source dataset in strips.

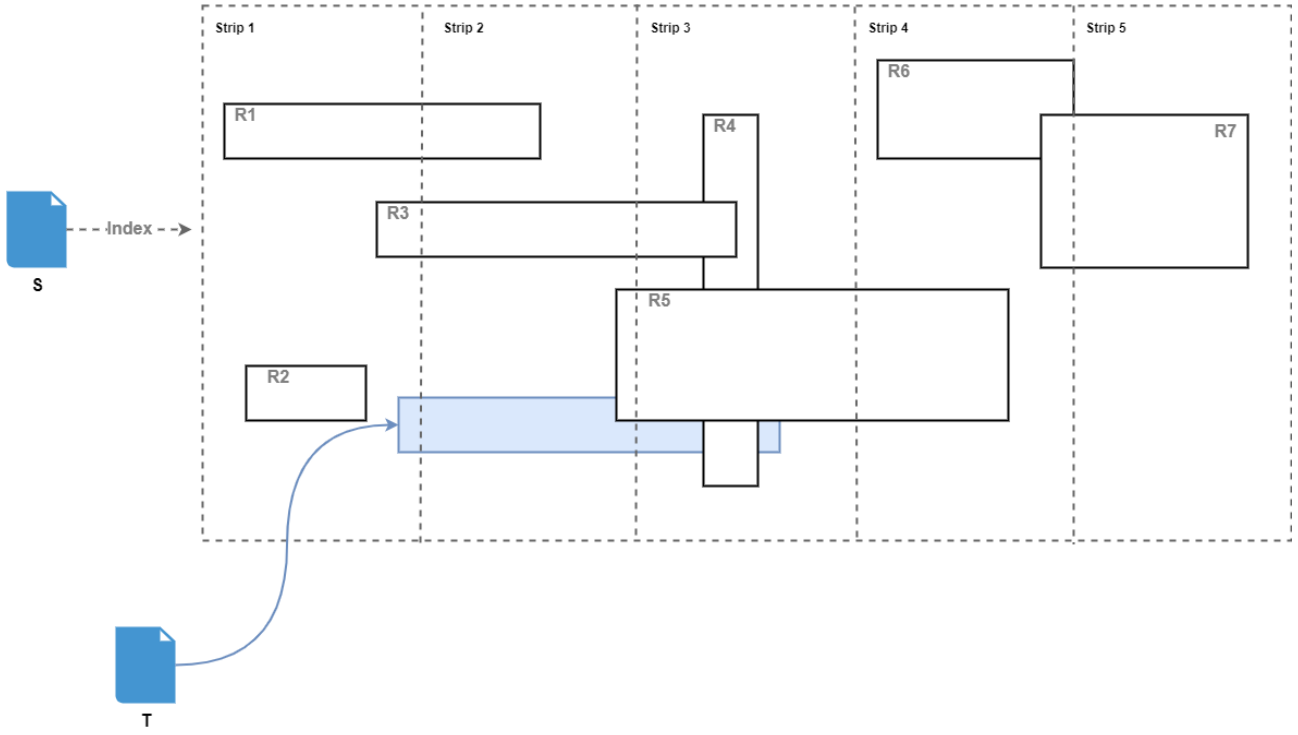

Figure 12. Strip Sweep: Find the strips intersecting the MBR of the target geometry.

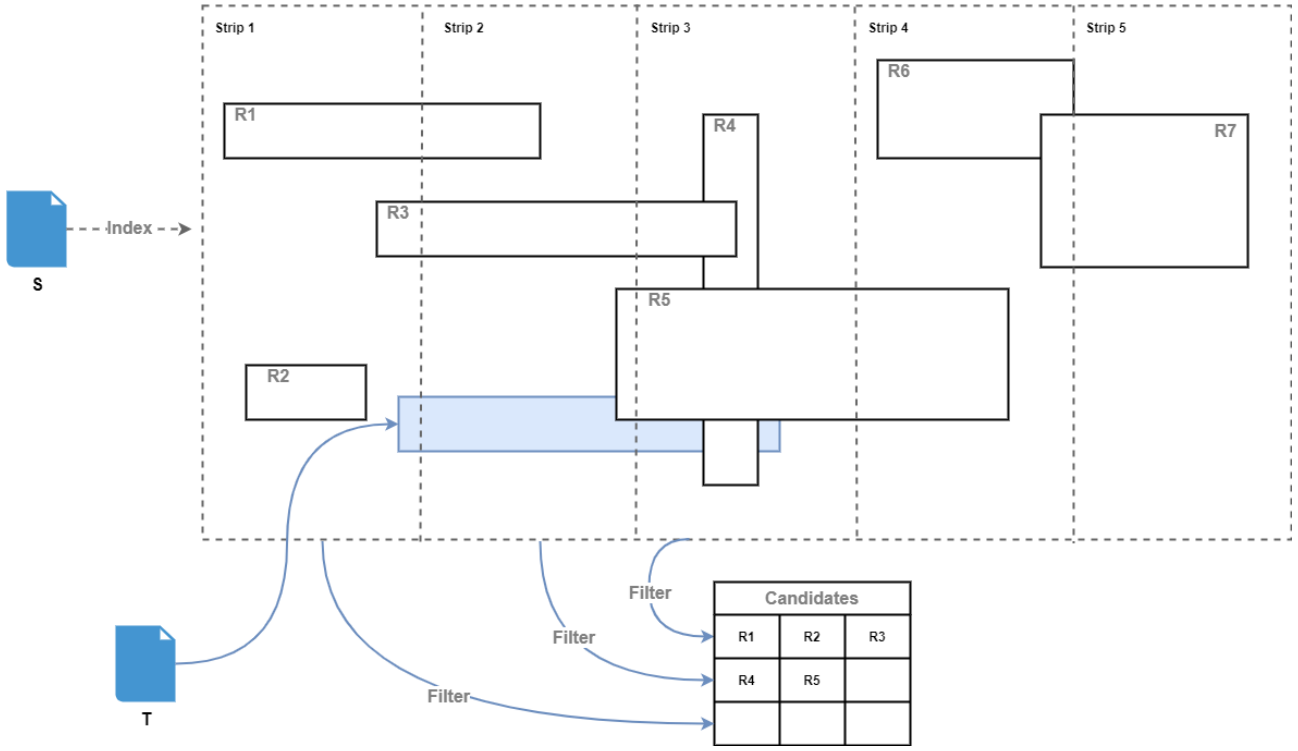

Figure 13. Strip Sweep: Add all the source geometries from the strips into the set of candidates.

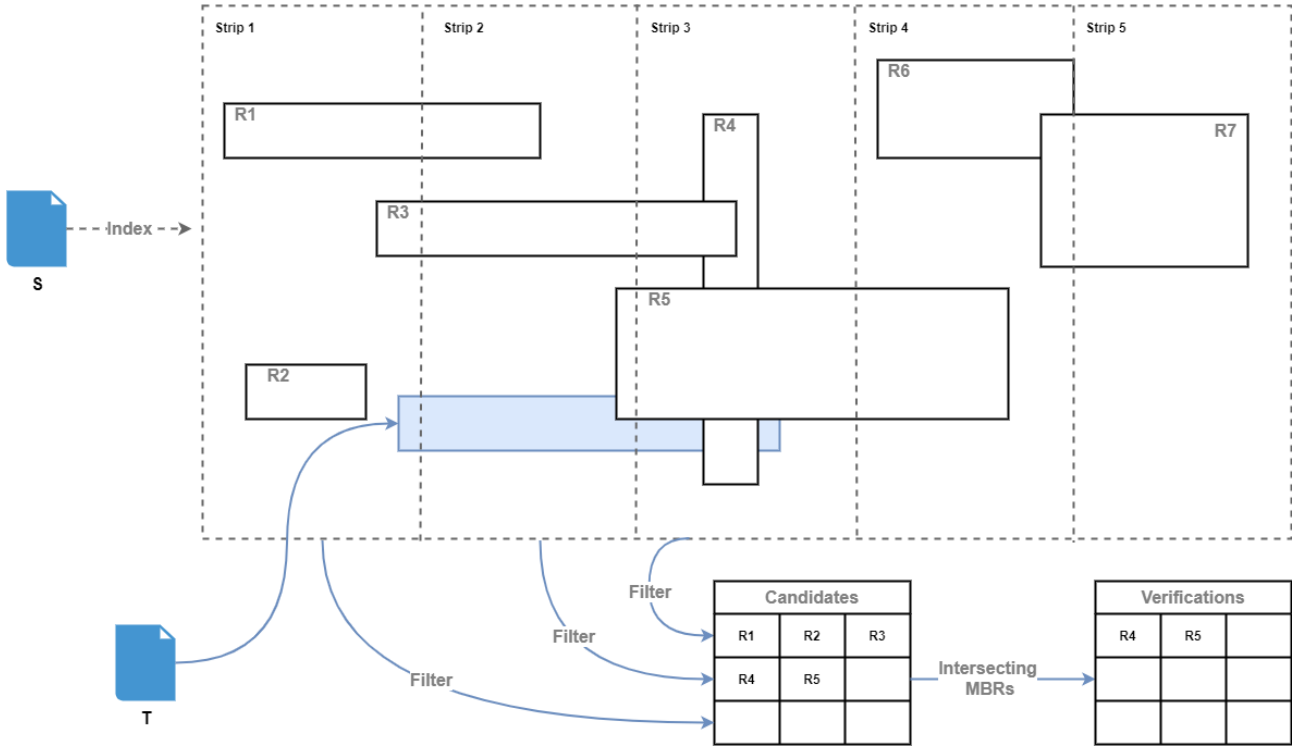

Figure 14. Strip Sweep: Verify the candidate geometries that have intersecting MBRs.

3.4. Strip Sort-Tile-Recursive (STR) Sweep

Considering all the geometries within the strip as candidates, imposes a significant amount of overhead. Taking into account a scenario, where a target geometry expands to half the strips, about half the geometries of the source dataset are regarded as candidates. This is better than using a cartesian join between the source and the target dataset, but still involves many irrelevant candidate geometries. Hence, instead of saving the geometries of each strip in a hashmap, we introduce an STR-Tree, which allows for low access time: probing into an STR-Tree has an average search complexity of $O(log_M n)$, where M is the number of entries of each node. Depending on the dataset, such a tree can result in very fast filtering time, almost $O(1)$, if its height is very small. This is the case with our implementation, which maintains a separate STR-Tree for each strip.

The STR-Tree implementation is provided by JTS and it is an R-Tree packing method based on the approach by Leutenegger et al. [16].

As mentioned, the only difference between *Strip Sweep* and *Strip STR Sweep* is the way the geometries are filtered: instead of considering all source geometries in the intersecting strips as candidates, this algorithm filters out those not overlapping with the target geometry on the vertical axis. This approach is illustrated in the following figures: after indexing the source geometries in Figure 15, the rest of the process is the same, as seen in Figure 16, Figure 17 and Figure 18, except that Figure 17 considers the y-axis to filter out the irrelevant source geometries.

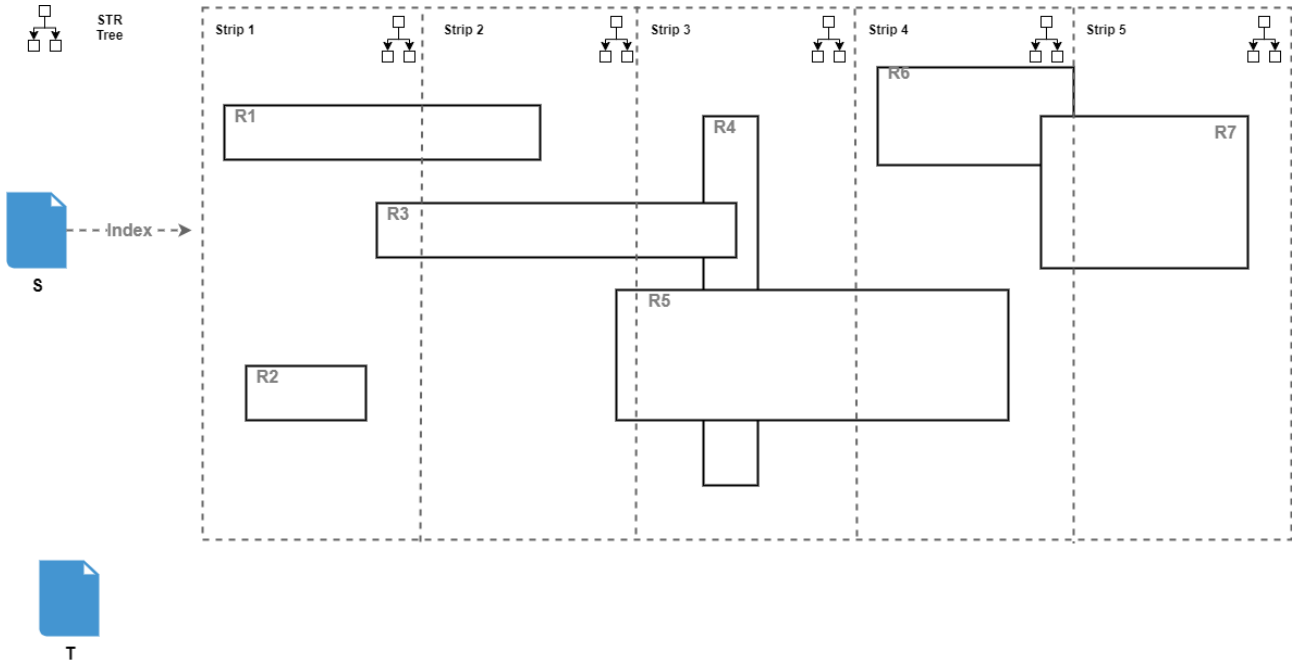

Figure 15. Strip STR Sweep: Partitioning the source dataset in strips indexed with an STR-Tree.

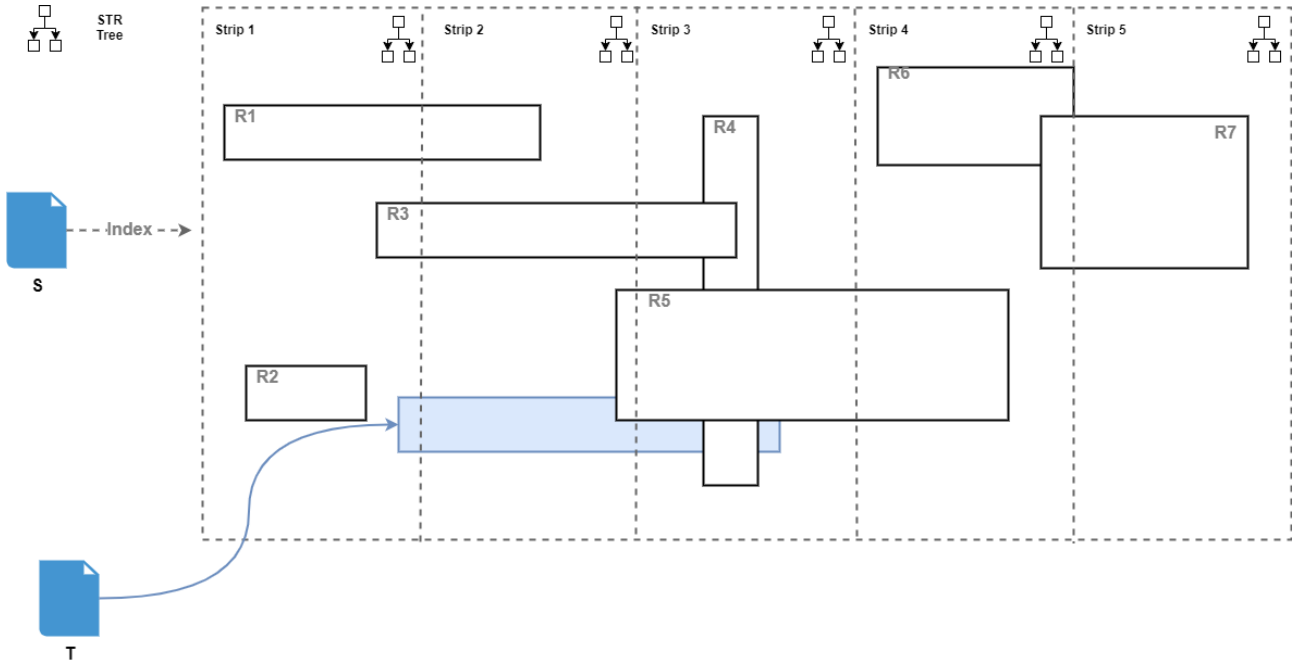

Figure 16. Strip STR Sweep: Find the strips intersecting the MBR of the target geometry.

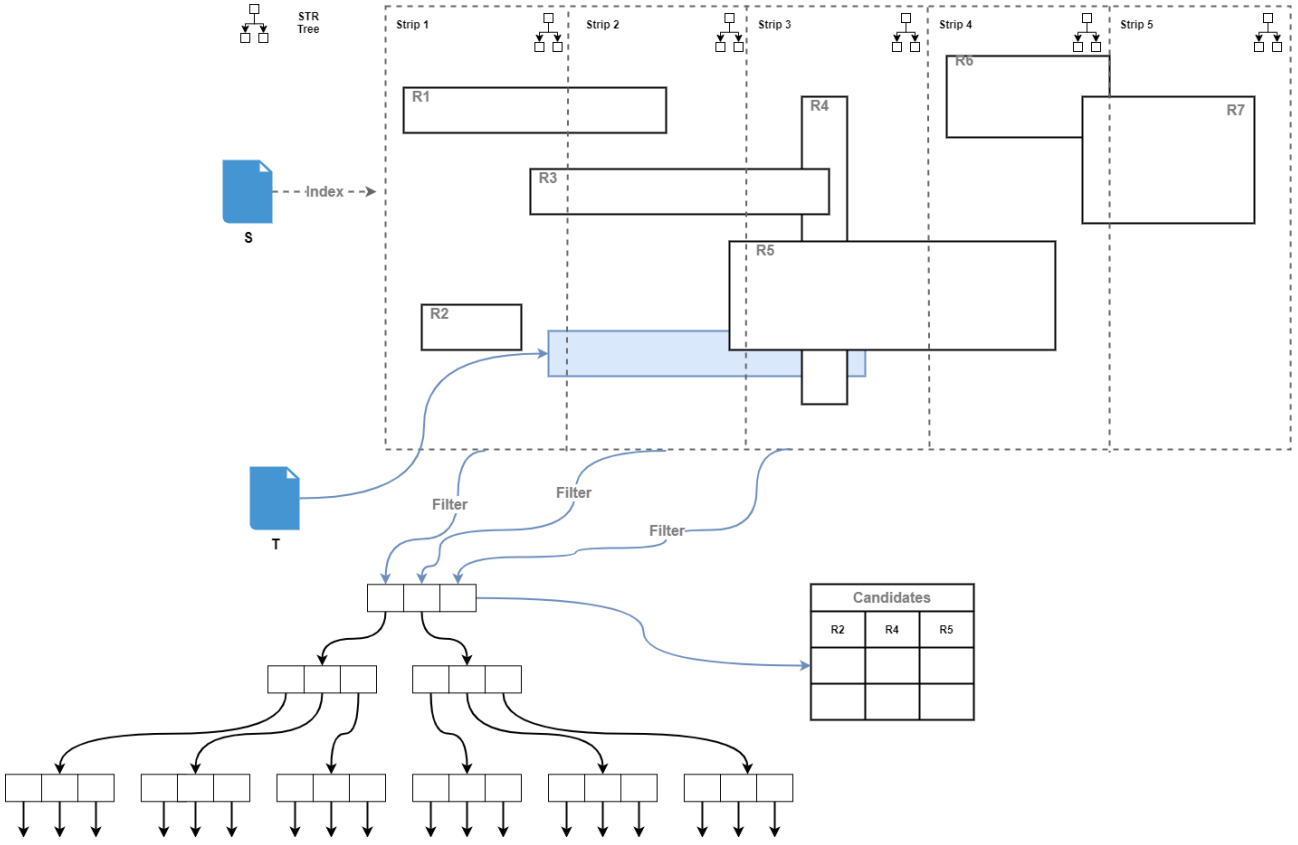

Figure 17. Strip STR Sweep: Filter the candidates in each strip using the respective STR-Tree.

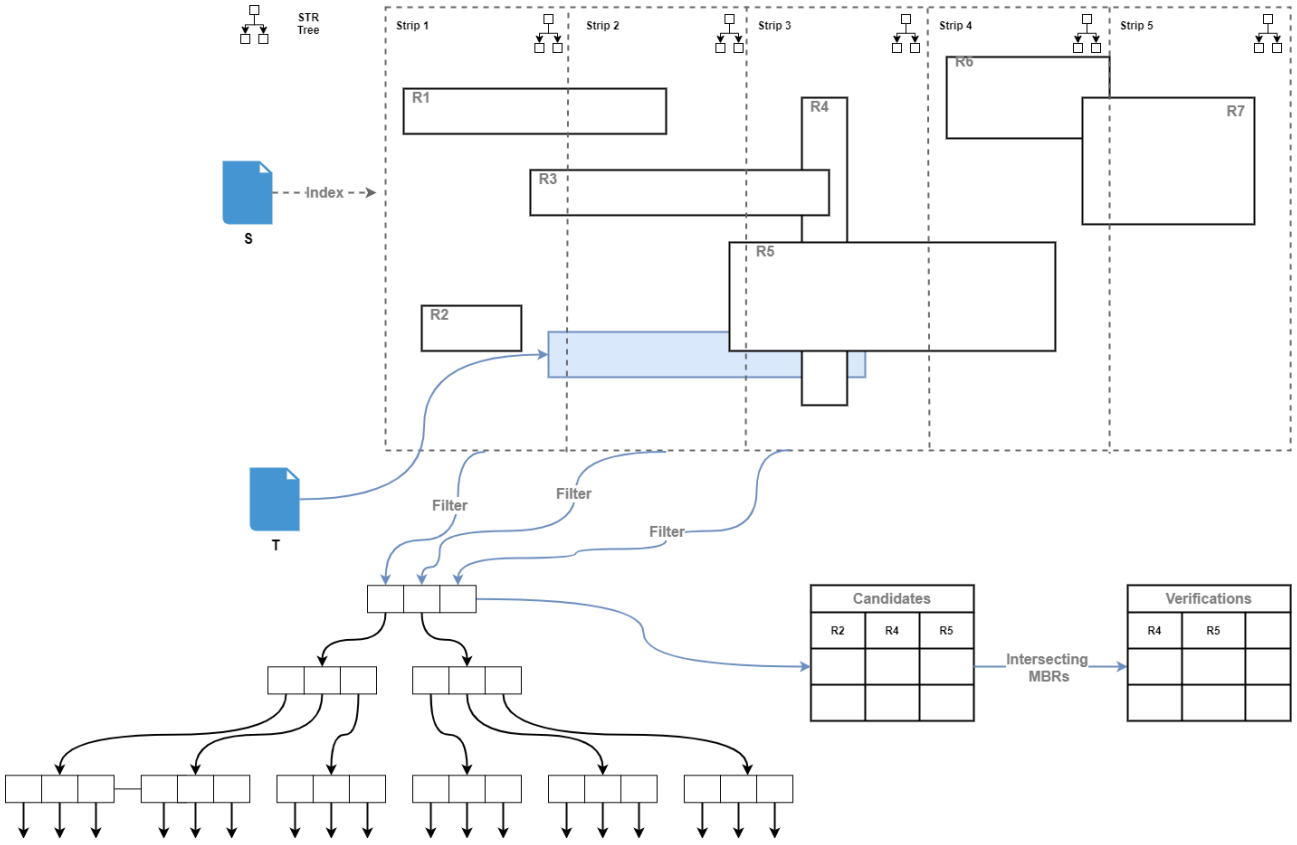

Figure 18. Strip STR Sweep: Verify the candidate geometries that have intersecting MBRs.

3.5. R-Tree Join

One of the most important spatial indexes is the *R-Tree* [12]. R-Tree was introduced by Antonin Guttman as a dynamic mechanism to index and retrieve spatial entities in a geodata database. It belongs to the family of height-balancing trees like the B-Tree. It is widely used in spatial databases, where each leaf node points to records on the disk. Non-leaf nodes contain pointers to child nodes and an MBR that encloses the span of all the MBRs in the child nodes.

The basic idea of the R-Tree is to recursively cluster M geometries using an MBR to envelope the area they expand. When inserting a new record to the spatial index, the record starts from the root and recursively chooses the subtree whose envelope needs the least expansion to include it, until it finds a leaf node with available space. In case it is added to a full leaf of M entries, the node needs to be split into two leaf nodes that contain $M + 1$ entries, in total. In the event of two leaf nodes not being able to fit in their parent node after the split, this operation may be propagated to their parent nodes resulting in an overall expansion,

Guttman provides two algorithms for splitting a node, a *Quadratic-Cost Algorithm* and a *Linear-Cost Algorithm*. JedAI-spatial uses the former one, where it picks the two largest geometries in order to initialize the two new nodes created after the split. Then, it assigns each remaining geometry to the node whose MBR expands the least after the insertion.

On average, the searching process for a particular record on the R-Tree has a complexity of $O(log_M n)$, where M is the maximum number of entries that fit in one node (i.e., node utilization). Starting from the root, a target MBR is following recursively the children sub-trees which intersect its MBR until the leaf nodes are reached. The disk accesses for this task, are equal to the height of the tree plus one for retrieving the leaf records from the

disk, hence $O(log(h+1))$. The smaller the height of the tree, the lower the disk access times.

Example 4. A simple R-Tree is illustrated in Figure 19 along with its memory representation in Figure 20.

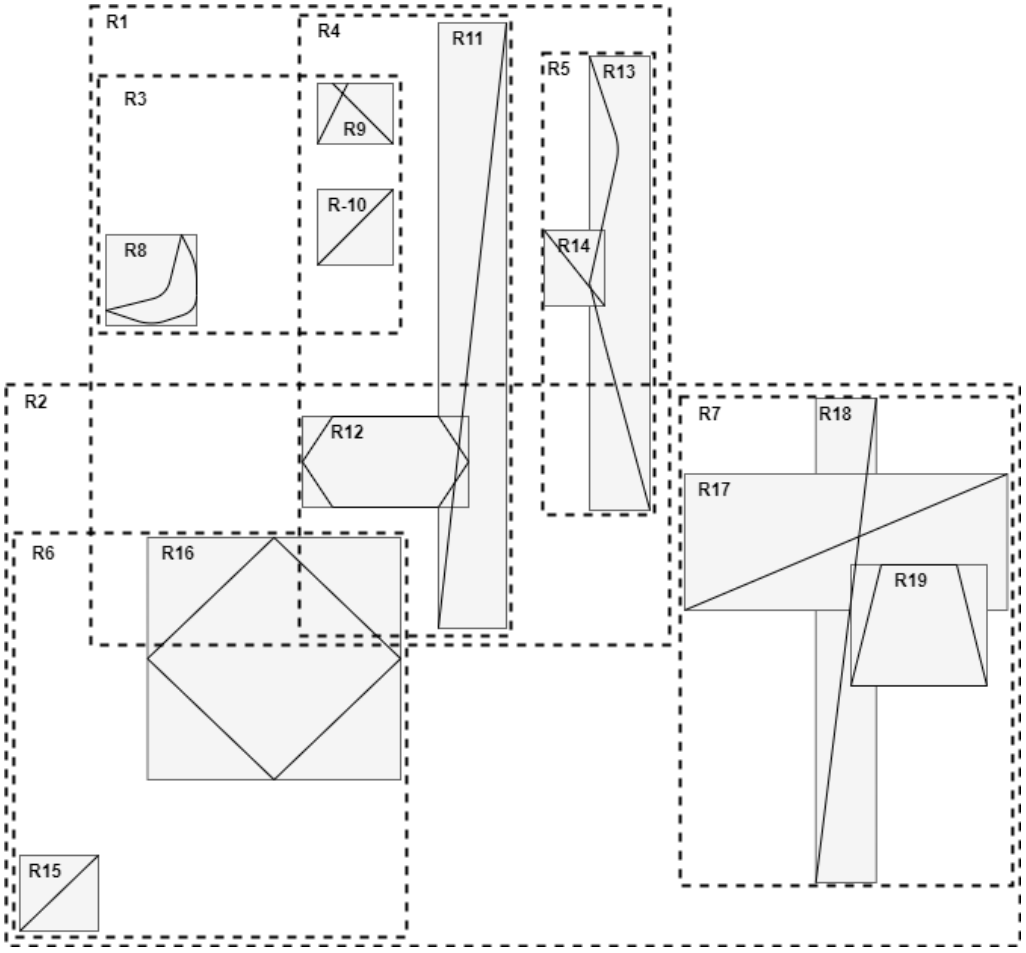

Figure 19. An example of R-Tree space indexing.

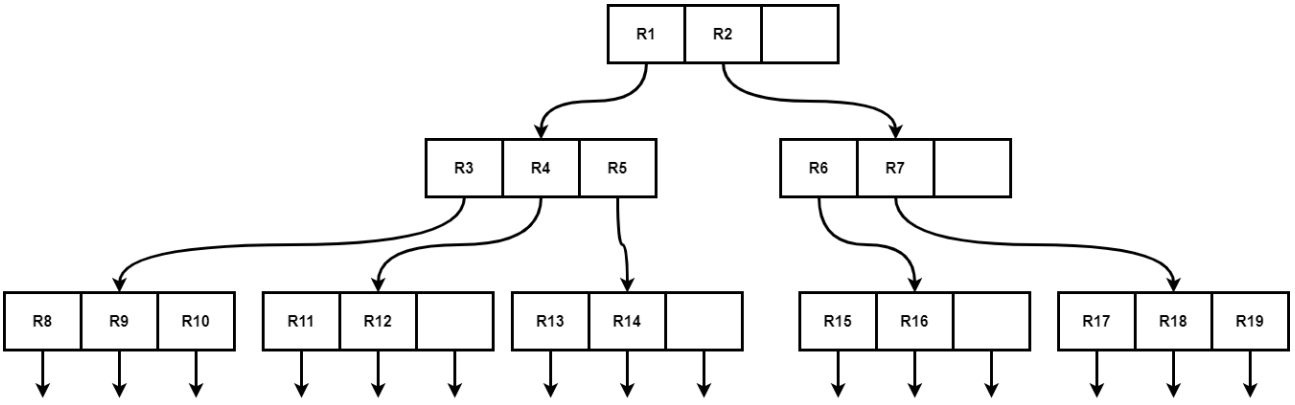

Figure 20. The R-Tree data structure.

JedAI-Spatial utilizes the R-Tree in order to index only the source dataset, so as to minimize the memory requirements. In the filtering step, each target geometry, which is read from the disk on the fly, is queried on the R-Tree Spatial Index to locate all possible candidates. These candidates are thereupon verified as long as their MBRs intersect with the target geometry's MBR.

Algorithm 4: R-Tree Spatial Join

Input: source dataset S, target dataset T
Output: set of links L with the detected topological relations
Index the source dataset using an R-Tree
while T *is not empty* **do**
  Let $t = Head(T)$
  Query the R-Tree Spatial index with t
  Let the query results S'
  while S' *is not empty* **do**
    Let $s' = Head(S')$
    if $s'.intersects(t)$ **then**
      $L+ = verifyRelations(s', t)$
    end
  end
end

Figure 21. R-Tree Spatial Join Algorithm.

Example 5. This algorithm is illustrated step-by-step in Figure 22 - Figure 25. Firstly, the source dataset is indexed using the R-Tree as a spatial index (Figure 22). Then, the current target geometry is queried on the R-Tree in Figure 23. All the leaf nodes whose parent's MBR intersects with that of the target geometry are regarded as candidates (Figure 24). Finally, the MBRs of all candidate source geometries are checked to ensure that they intersect with the MBR of the target geometry. If they are, their relations are verified (Figure 25).

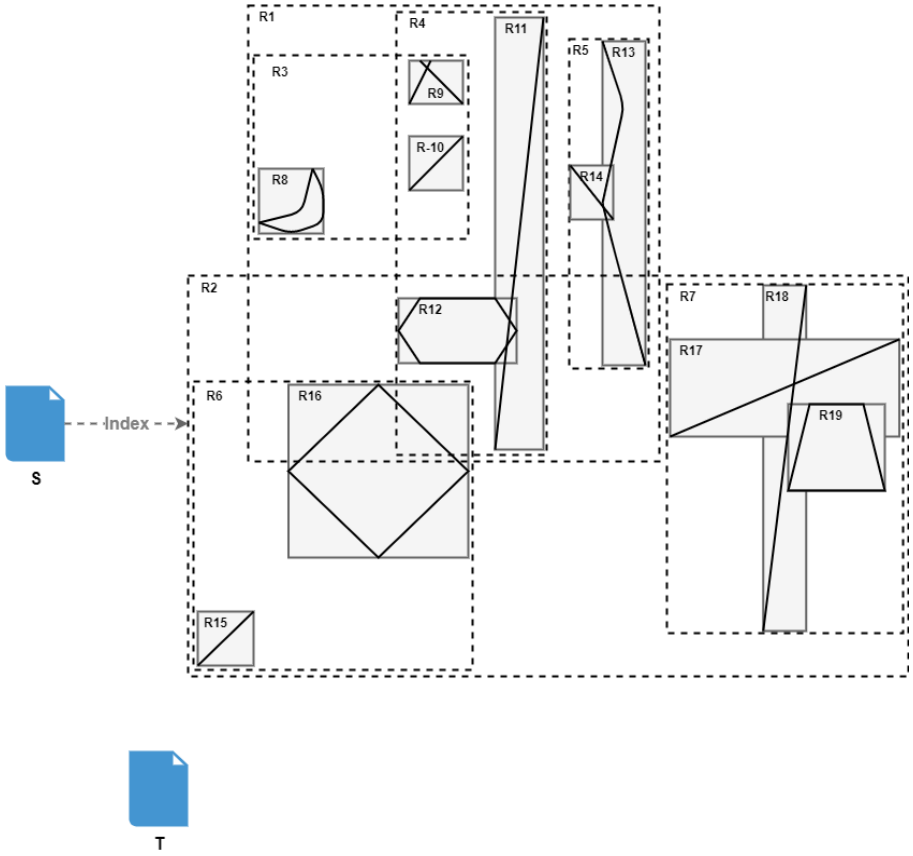

Figure 22. R-Tree: Indexing the source dataset.

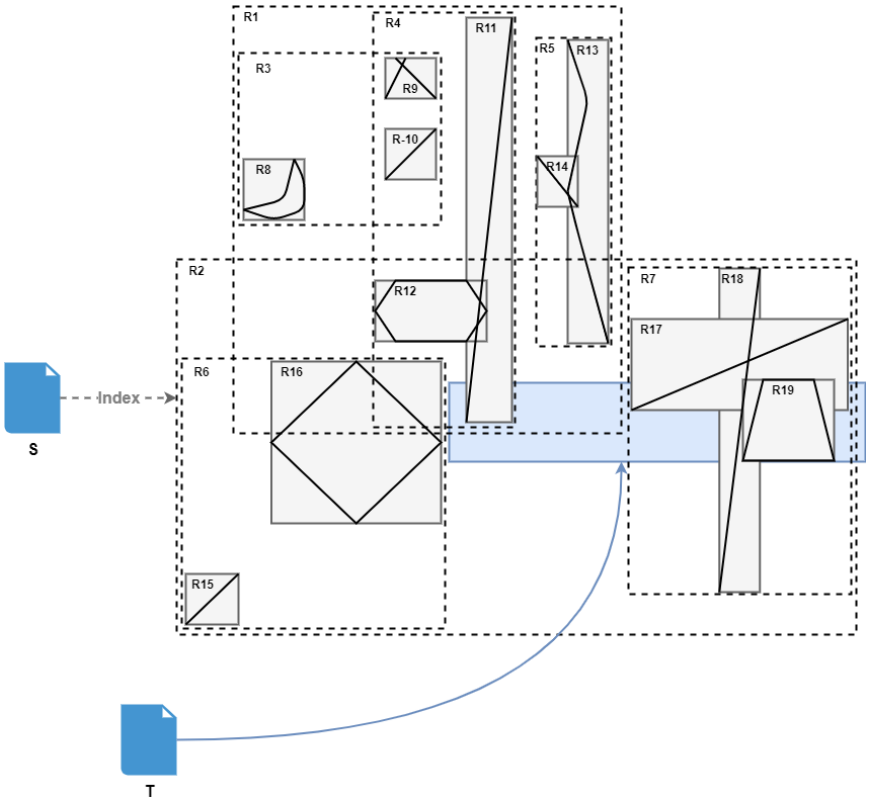

Figure 23. R-Tree: Find the nodes the target geometry resides in.

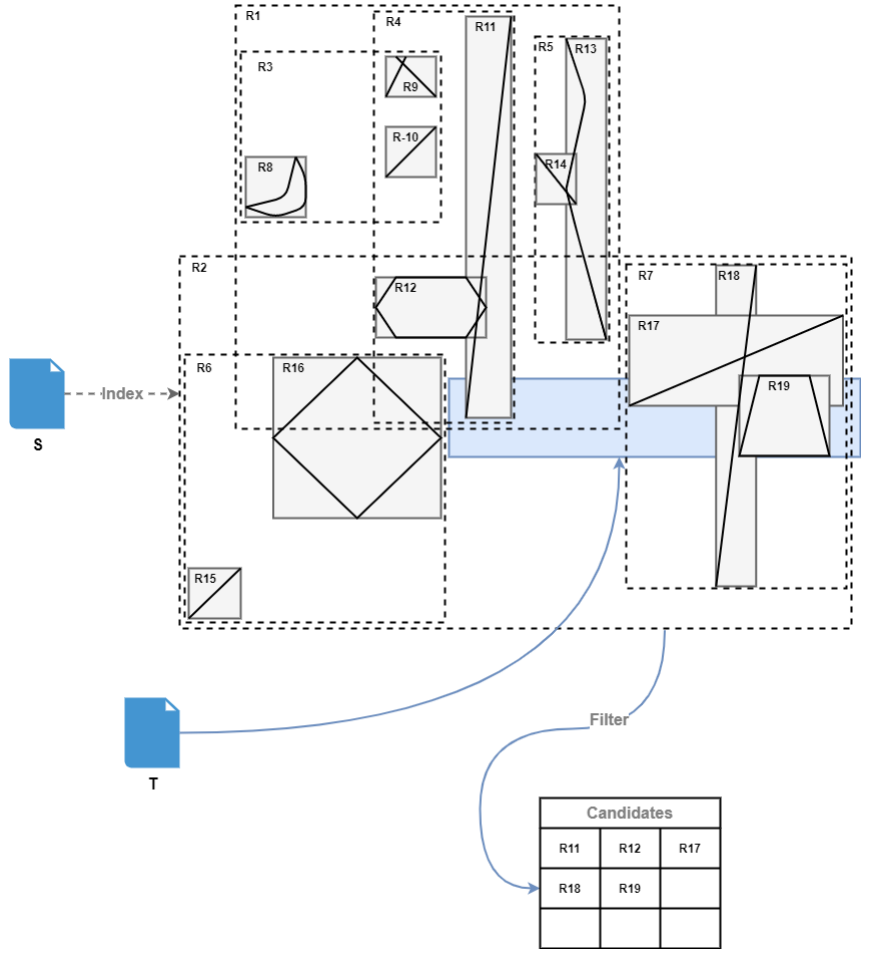

Figure 24. R-Tree: Each geometry belonging to the selected nodes is a candidate.

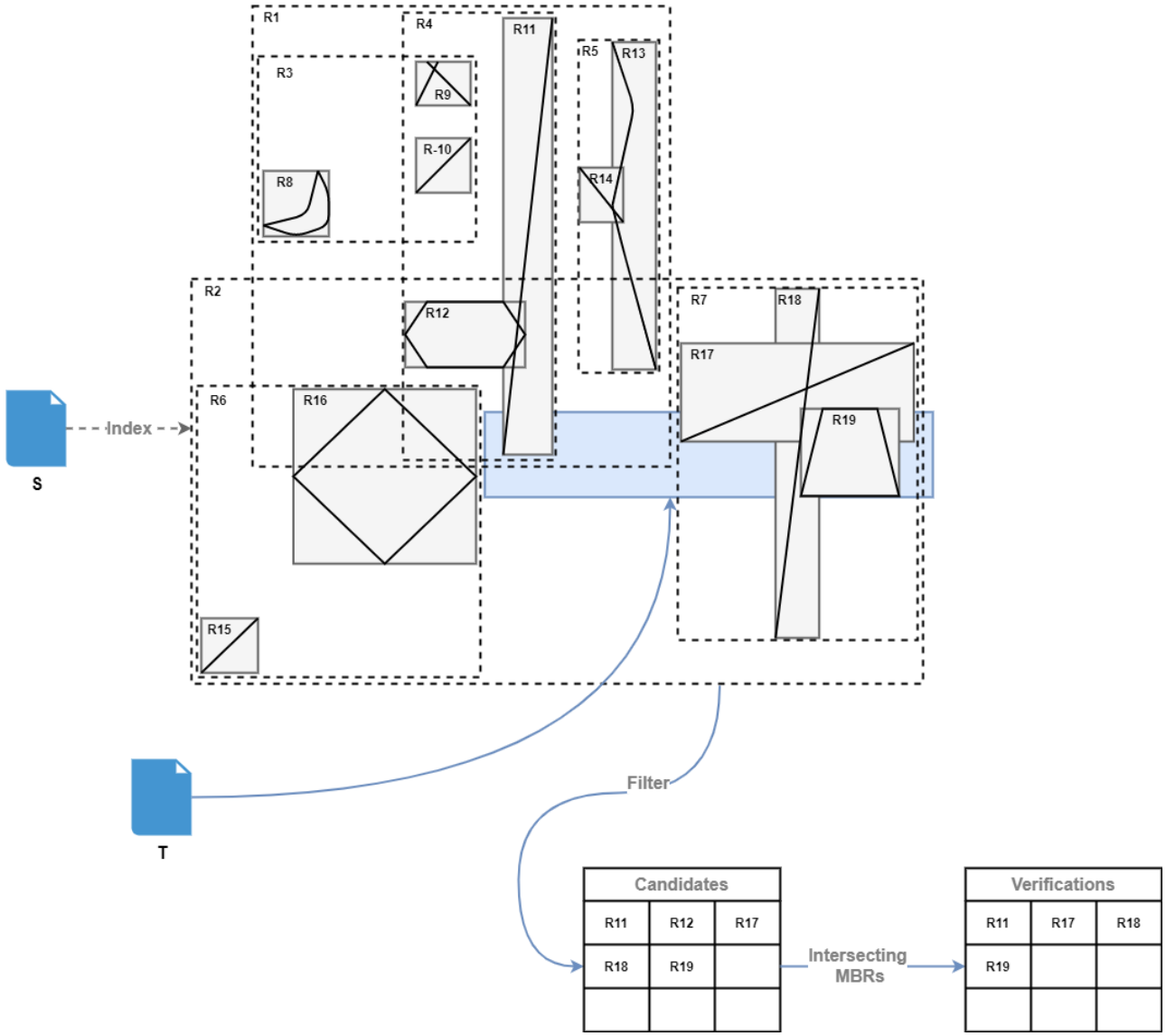

Figure 25. R-Tree: Ensure that for each source candidate, its MBR intersects with the MBR of the target geometry.

3.6. Cache-Conscious R-Tree (CR-Tree) Join

Technology is becoming prevalent and commercial, resulting in continuing price dropping and an overall growth in specifications of tech related gear. Thus, memory is not only getting cheaper but also faster and larger.

Cache-Conscious R-Tree (CR-Tree) introduced by [13] aims to leverage the L1 and L2 cache memory so as to have faster access times. Its basic operations (insert, search, delete) are identical with that of R-Tree; the only difference is that a new technique is proposed for reducing the size of MBRs and hence, cache-misses, called *Quantized Relative Representation of MBR (QRMBR)*. This technique is the result of transforming the children's MBRs based on their parent's MBR relative position, known as *Relative MBR (RMBR*). This process is illustrated in Figure 26.

The QRMBR technique is based on the following formula, where C is the original MBR, l the quantization level and I the reference MBR:

$$QRMBR_{I,l}(C) = (\varphi_{I.x\,\min,\,Ix\,\max,\,l}(C.xmin),\ \varphi_{I.y\,\min,\,Iy\,\max,\,l}(C.ymin)$$
$$\Phi_{I.x\,\min,\,Ix\,\max,\,l}(C.xmax),\ \Phi_{I.y\,\min,\,Iy\,\max,\,l}(C.ymax))$$

$$\text{where } \varphi_{a,\,b,\,l}(r) = \begin{cases} 0 & , if\ r \le a \\ l-1 & , if\ r \ge b \\ \lfloor l(r-a)/(b-a) \rfloor & , otherwise \end{cases}$$

$$\text{and } \Phi_{a,\,b,\,l}(r) = \begin{cases} 1 & , if\ r \le a \\ l & , if\ r \ge b \\ \lceil l(r-a)/(b-a) \rceil & , otherwise \end{cases}$$

Example 6. Taking R_1, its RMBR's lower left corner is $(3,\ 7)$ and top right corner $(10,\ 15)$. This results from subtracting R_1's original lower left corner $(43153,\ 27087)$ with the lower left corner of R_0, $(43150,\ 27080)$. The same applies to their top right corners. For a QRMBR with a quantization level of $l = 16$, using the above formula, R_1's envelope is defined as: $QRMBR_1 = (xmin,\ ymin,\ xmax,\ ymax) = (1.5,\ 3,\ 5,\ 7.5) = (1, 3, 5, 8)$.

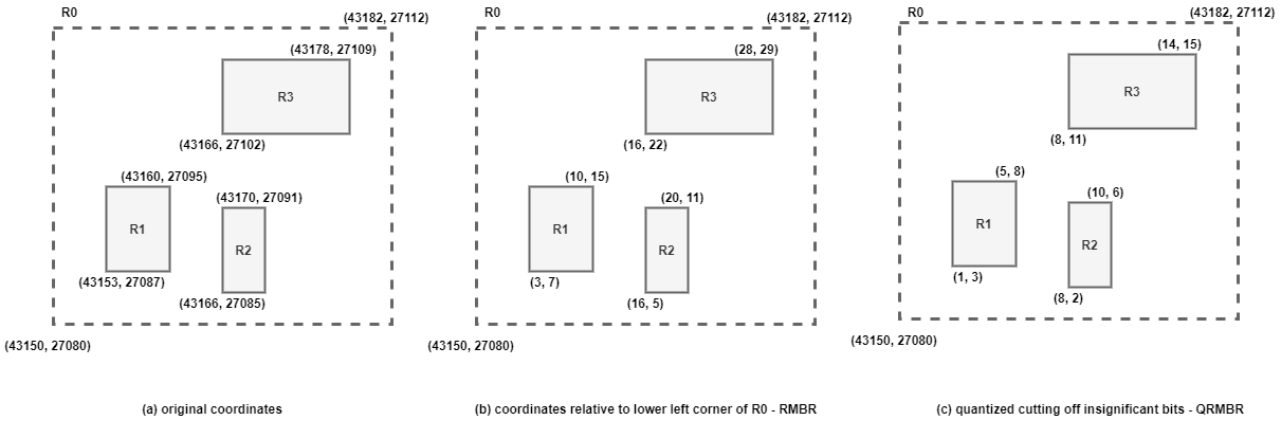

Figure 26. The QRMBR method.

This approach significantly compresses the size of the MBRs which take up most of the space of an R-Tree structure. According to the authors, a CR-Tree is wider and smaller compared to an R-Tree, increasing the performance and occupying about 60% less memory. The overall overhead of compressing the MBRs is less than the overhead inflicted by cache-misses.

The algorithm is similar to the one in R-Tree Spatial Join in Figure 21, with the only difference that the spatial index used is the CR-Tree.

3.7. Quad Tree Join

The last spatial join sequential method implemented under the scope of this thesis is the Quad Tree Spatial Join. Quad Tree [14] is another state-of-the-art spatial index for storing and retrieving geodata. It is regarded as an enhanced version of a binary tree specialized to care for 2-dimensional data. Each node's out-degree is equal to four, meaning it can have up to four children. As a result, the children of a node divide the space into four quadrants: NorthEast (NE), NorthWest (NW), SouthEast (SE) and SouthWest (SW). See Figure 27 for an example. The insertion complexity is $O(logn)$, where n the data size.

The Quad Tree implementation used for the spatial indexing in the experiments is the one by JTS. Every cell has a maximum capacity M. When M is reached, the corresponding cell is split into four new ones, its children. The insertion algorithm is similar to binary trees. When inserting a new entry in the spatial index, starting from the root, the geometry recursively chooses the quadrant/subtree which covers its MBR, until it reaches a leaf node, whereby it is saved.

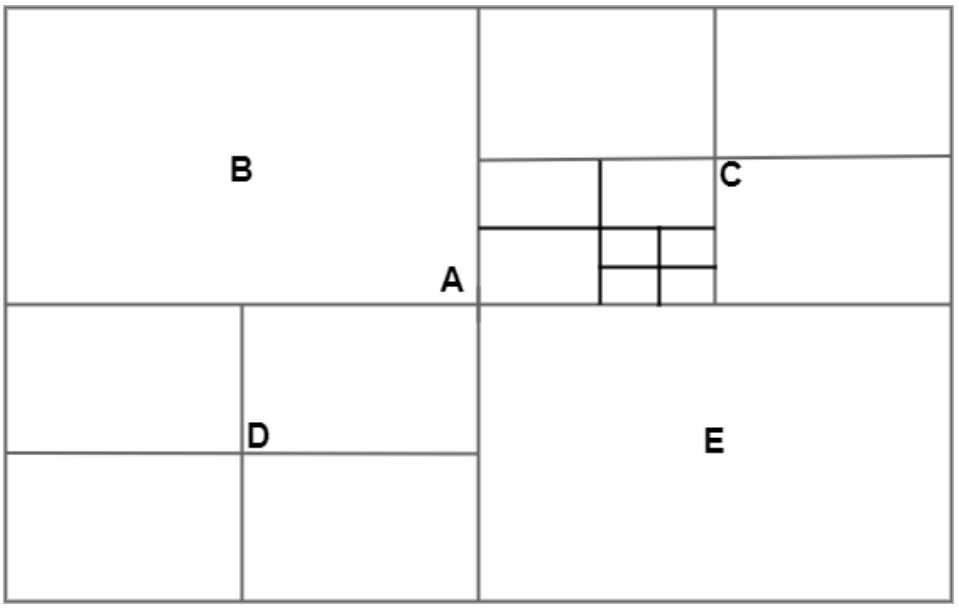

Figure 27. The division of space using a Quad Tree. Starting from root A, the space is divided into four quadrants B, C, D and E. Each quadrant now can be divided even further to another four quadrants.

The Quad Tree Spatial Join algorithm is a lot similar to the previous tree-based algorithms, the R-Tree and CR-Tree Spatial Join. The source dataset is indexed using a Quad Tree and every geometry of the target dataset is probed on the Quad Tree. The whole process is outlined in Figure 21.

3.8. Grid-based Algorithms

They index the input geometries 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 4 state-of-the-art algorithms of this type, which differ in the definition and use of the EquiGrid during Filtering and Verification.

RADON [17]. 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 across all geometries in S and T. Every geometry is placed in every tile that intersects its MBR. Verification computes the Intersection Matrix for all candidate pairs [29], taking special care to avoid the ones repeated across different tiles of the EquiGrid.

GIA.nt [18]. Filtering loads into main memory only the source dataset, i.e., the smallest one in terms of the number of geometries. The granularity of the EquiGrid index is determined by the average dimensions of the source geometries. Verification reads the target geometries from the disk, one by one; for each $t \epsilon T$, it gathers the set of candidate source geometries, i.e., the ones whose MBR intersects the same tiles as the MBR of t and computes the corresponding Intersection Matrix, adding the detected links to output L.

Static variants. Unlike the dynamic EquiGrid of the above algorithms, whose granularity depends on the input data, *Silk-spatial* [19] employs a *static* EquiGrid, whose granularity is predetermined, independently of the input characteristics. Even though the resulting index might be too fine- or coarse-grained for the input datasets, this approach is based on the idea that the resulting candidate pairs are eventually reduced by the requirement for intersecting MBRs. To put this approach into practice, JedAI-spatial includes **Static RADON** and **Static GIA.nt**, where the index granularity is manually defined.

Implementation improvements. RADON's implementation is publicly available through LIMES[15]. However, we re-implemented it in JedAI-spatial so as to significantly improve its performance. First, we reduce the run-time of Filtering by skipping the swapping strategy, which is used to identify the input dataset with the smallest overall volume. This has no impact on its functionality, given that the index granularity considers both input datasets. Second, we reduce RADON's memory footprint to a significant extent. Instead of the hashmap that stores all verified pairs in main memory to avoid verifying the same candidate pairs more than once, we use the *reference point technique* [9], verifying every candidate pair only in the tile that contains the top left corner of their intersection (see Figure 6). Moreover, unlike the original implementation, which refers to all references by their URL (of type `String`), we use ids for this purpose (of type `int`). We also use the data structures provided by the GNU Trove library[16], which work with primitive data types (e.g., the 4-bytes `int` instead of the 16-bytes `Integer`).

For GIA.nt, we use the open-source implementation[17] provided by the authors of [18], which already involves the memory footprint optimizations discussed above.

3.9 Budget-aware Algorithms

As explained above, these algorithms operate in a pay-as-you-go manner that aims to process as many related geometry pairs as possible within the limited available resources. These progressive algorithms receive as input the source and target datasets, S & T, as well as a budget BU, which specifies the maximum number of verifications that will be carried out. Their goal is to maximize the number of related geometry pairs that are detected after consuming the available budget. To this end, they follow the three-step framework in Figure 28. 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 in a way that places the likely related ones 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.

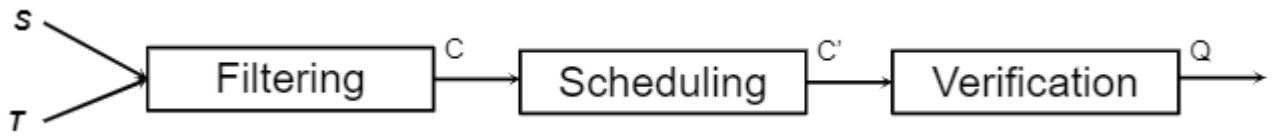

Figure 28. Three-step framework.

The gist of budget-aware algorithms is the combination of Scheduling with Filtering, as Verification is common to all algorithms. Based on the co-occurrence patterns of Filtering, Scheduling assigns a score to every *valid pair* of candidates, i.e., candidates with intersecting MBRs. The higher this score is, the more likely the constituent geometries are

[15] https://github.com/dice-group/LIMES
[16] http://trove4j.sourceforge.net/html/overview.html
[17] https://github.com/giantInterlinking/prGIAnt

to satisfy at least one topological relation. JedAI-spatial offers the following weighting schemes:

1. Co-occurrence Frequency (CF) measures how many tiles intersect both the MBR of the source and the target geometry.
2. Jaccard Similarity (JS) normalizes (CF) by the number of tiles intersecting each geometry.
3. Pearson's x^2 test extends CF by assessing whether the given geometries appear independently in the set of tiles.
4. Minimum Bounding Rectangle Overlap (MBRO) returns the normalized overlap of the MBRs of the two geometries.
5. Inverse Sum of Points (ISP) amounts to the inverse sum of boundary points in the two geometries, thus promoting the simpler candidate pairs.

Below, we explain how these weighting schemes are leveraged by each progressive algorithm.

Progressive GIA.nt [18]. It applies the same Filtering as its budget-agnostic counterpart and, then, its Scheduling retains the top-BU weighted valid candidate pairs.

Progressive RADON [18]. 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. Inside every tile, it identifies the non-redundant candidate pairs using the reference point technique. Those that are valid, too, are processed in decreasing score, as determined by the selected weighting scheme.

Iterative Algorithm. The above algorithms might exclude some geometries completely from the BU retained candidate pairs. To avoid this, we implemented a new progressive algorithm that applies the same Filtering as GIA.nt and its Scheduling retains BU/|T| candidates per target geometry. The top-weighted BU geometries are then forwarded to Verification.

Geometry-ordered Algorithm. This is another new progressive algorithm that assumes that the larger the average weight of a geometry is, the more likely it is to be related to its candidates (e.g., a road that touches a lot of buildings). Thus, it applies the same Filtering as GIA.nt and then, it estimates the average weight per target or source geometry. After sorting the geometries in decreasing average weight, it iterates once more over the target dataset to select the BU retained pairs from the candidates of the top-weighted geometries. Optionally, the retained candidates can be sorted in decreasing weight.

Note, though, that the analytical examination of the relative performance of these algorithms lies out of the scope of this thesis. We described the progressive methods supported by JedAI-spatial for the sake of completeness.

4. PARALLEL IMPLEMENTATIONS

The parallel algorithms were implemented on top of Apache Spark [8], a scalable, optimal, fault-tolerant, in-memory map-reduce framework implemented in Scala. Apache Spark employs a read-only collection of objects called *RDD* (Resilient Distributed Dataset). RDDs distribute the data into partitions, which are stored and processed in different nodes across the cluster. This abstraction allows developers to engineer software without worrying about the internal procedures regarding Spark, such as rebuilding a partition that was lost during execution or synchronizing all the workers.

The way the Spark cluster environment works is outlined in Figure 29. The process is the following:

1. The Driver Program instantiates a SparkContext instance, which acts as the master in the whole process.
2. The Driver Program connects to the Cluster Manager (i.e. Spark Standalone, YARN), which allocates resources based on the SparkContext established in the first step. Then, the latter launches the executors requested.
3. Spark acquires the executors running in each Worker Node and sends the source code.
4. The Driver Program assigns tasks (map) and collects them when they are done (reduce).

Apache Spark is the immediate successor to Apache Hadoop. Their main difference is that Spark carries out its processes on the main memory instead of the hard disk. However, if the memory size is not adequate, it utilizes the hard disk as well.

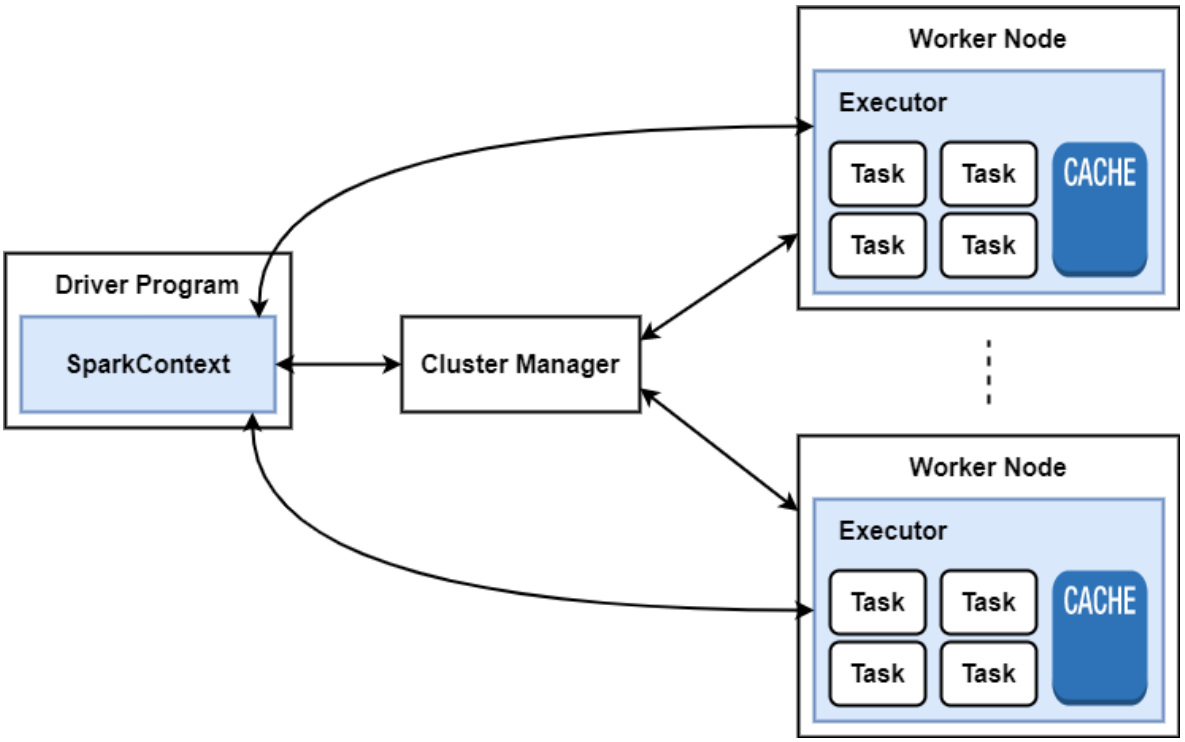

Figure 29. Apache Spark Cluster Overview.

As already mentioned, the selection and implementation of the parallel algorithms incorporated in JedAI-spatial was based on Pandey et al [7]. This paper compares different Hadoop and Spark systems and frameworks through a thorough evaluation. JedAI-spatial integrates all the Apache Spark based systems excluding SIMBA, due to the fact that it does not support other data types besides points.

JedAI-spatial can be considered as a level above the actual frameworks, since the developers are required to solely provide a YAML[18] configuration file denoting the partitioning algorithm, the spatial index to be used, the source and target dataset file paths as well as the number of partitions. The configurations for each parallel method change. More information can be found on the ANNEX I.

Each parallel implementation is divided in three consecutive phases:

1. Preprocessing Stage
2. Global Join Stage
3. Local Join Stage

The **Preprocessing Stage** prepares the data for the main processing phases. First of all, the source and target datasets are read from the HDFS in order to be transformed into RDDs. Then, they are split into logical/physical partitions based on a partitioning method. This is a classic technique in Apache Spark. Remember that it is a distributed framework running on a cluster. The main idea is to group the data into partitions so as to reduce the Spark shuffles and hence increase the performance. A Spark shuffle is when data is rearranged between partitions.

In more detail, each framework uses its own set of partitioning techniques, such as Quad Trees or Z-Order Curves, in order to partition the source data. Then, the geometries of both source and target RDD datasets are assigned an identifier based on the partitions they lie in space. The RDD is shaped as $< K, V >$ where K is the identifier of the partition and V the geometry. In case a geometry belongs to multiple partitions, it is assigned multiple identifiers, i.e. $< K1, V >, < K2, V >$.

In the **Global Join Stage**, the source and target RDDs are joined in the same partitions based on their key K, the partition identifier. Each partition is a processing unit in the cluster. The joined RDD is shaped as $< K, V >$, with V changing to a tuple of $(Iterable[SourceGeometries], Iterable[TargetGeometries])$.

The third and final phase, **Local Join Stage**, performs the spatial join on the candidate geometries within the same partition. Two are the available techniques, each having two steps:

1. **Nested Loop Index Join**:
    a. Filtering Step: Identifies all intersecting MBRs between target and source candidates using a Spatial Index built from the $Iterable[SourceGeometries]$.
    b. Verification Step: Verify the topological relations of each pair of source-target candidates.
2. **Nested Loop Join**:
    a. Filtering Step: Filter all the intersecting MBRs between target and source candidates by comparing each target from $Iterable[TargetGeometries]$ with each source geometry from $Iterable[SourceGeometries]$.
    b. Verification Step: Verify the topological relations of each pair of source-target candidates.

Integrating the parallel implementations was a challenging task. Each framework uses a different Spark version, libraries as well as philosophy. However, all of them were open-source, which bestowed major guidance.

JedAI-spatial focuses on migrating only the spatial join process of the above implementations to a central repository. This meant migrating all the libraries and data

[18] YAML stands for Yet Another Markup Language.

structures in accordance with the Java Topology Suite (JTS) to ensure compatibility and uniformity. The reasoning behind this choice was based on three factors:

1. Easier code maintenance and bug fixing. Some frameworks appeared to not be working properly and almost all have not received an update in a very long time. It was mandatory to keep the dependencies up-to-date.
2. Upload JedAI-spatial as a web application or a Maven/Gradle dependency.
3. Have it open-source so anyone can contribute given the proper guidelines.

As already mentioned, JedAI-spatial also employs the Reference Point Method for faster filtering. This method is far superior to using Spark `distinct` for the duplicate elimination, because the latter shuffles the data along the cluster imposing massive overhead. For more details, please refer to the Experimental Results section.

Below, we present the parallel frameworks integrated within JedAI-spatial as well as the improvements we incorporated in order to enhance their functionality.

4.1. Spatial Spark

Spatial Spark [20] is an open-source framework for spatial query processing which emerged as a pioneer for extending Apache Spark and Cloud Impala with spatial querying for distributed processing. It consists of two methods: a *broadcast spatial join* and a *partitioned spatial join,* which aimed to address the problems that arose in the broadcast join due to a limitation in serializing (see below for more details). The source code of Spatial Spark is uploaded on GitHub[19].

4.1.1 Spatial Spark Broadcast Join

This method embodies the initial approach implemented by the authors which leverages Spark's *Broadcast Variables*. Sparks allows engineers to broadcast a variable as read-only to the whole cluster, which is then saved to each worker. In that way, developers can use a shared data structure in all computing nodes.

4.1.1.1 Process

In particular, consider a source and a target dataset. Spatial Spark indexes the target dataset using an R-Tree, which is then broadcast to all the worker nodes. Each worker receives the spatial index of the target dataset, iterates over the source geometries and filters all the potential candidates using the broadcasted R-Tree. After checking if the candidates' MBRs intersect, it discovers all their topological relations. This whole process is illustrated in Figure 30. It lacks the Global Join Stage, because the datasets are not merged in any way.

[19] https://github.com/syoummer/SpatialSpark

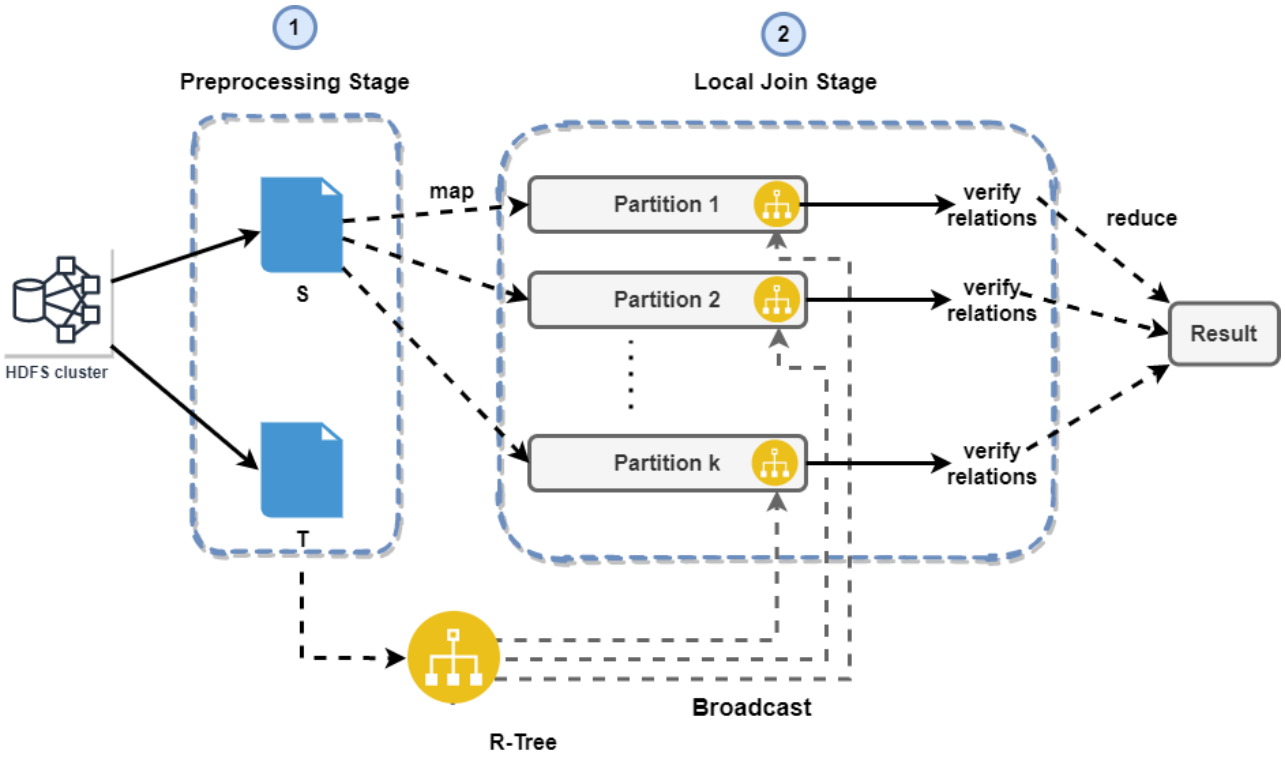

Figure 30. JedAI-spatial's Spatial Spark Broadcast Join.

This technique is simple and adequate for small datasets. Especially small target datasets that are less than 2GB. JedAI-spatial uses *Spark 2.4.5* and the maximum value of KryoSerializer's buffer is 2048MB. *KryoSerializer* is an optimal serializer for Apache Spark that is responsible for the serialization of RDDs among the computing nodes. As long as the R-Tree constructed on the target dataset is less than the indicated serializing limit, Spatial Spark's Broadcast Join can run without any problems.

4.1.2. Spatial Spark Partitioned Join

To overcome the size limit of the Broadcast Join implementation, Spatial Spark authors introduced a Partitioned Join variant. This method offers three partitioning techniques:

- *Binary Split Partition*: It samples a user defined ratio of geometries and recursively divides the extent of the source and target datasets into partitions until the level of the binary tree is equal to the level the user gave as input.
- *Sort Tile Partitions*: It samples a user defined ratio of geometries and applies the Sort Tile Recursive (STR) packing algorithm.
- *Fixed Grid Partition*: It divides the extent of the source and target datasets into $dimX \; x \; dimY$ partitions.

For the first two techniques, Spatial Spark samples the dataset with a ratio given by the user and applies the partitioning technique to the sample. The partitions are then inserted into an R-Tree structure. For the Fixed Grid Partition, there is no sampling; its purpose is to divide the entire space of all input data based on the dimensions given by the user as input.

4.1.2.1. Spatial Spark vs Spatial Spark in JedAI-spatial

The differences between the original Spatial Spark and the one on JedAI-spatial are listed in Table 2, with the most notable being the replacement of `distinct` with the reference point method. This enhancement was only applied to the partitioned join variant, since broadcast join does not join the data into partitions whatsoever.

Table 2. Spatial Spark vs JedAI-spatial's Spatial Spark.

	Spatial Spark	Spatial Spark in JedAI-spatial
Queries	Range, Spatial Join	Spatial Join
Duplicate Elimination	`distinct`	reference point technique

4.1.2.2. Process

Spatial Spark's Partition Join process follows the standard guidelines implemented in JedAI-Spatial. It is composed of the three stages declared at the onset of the section.

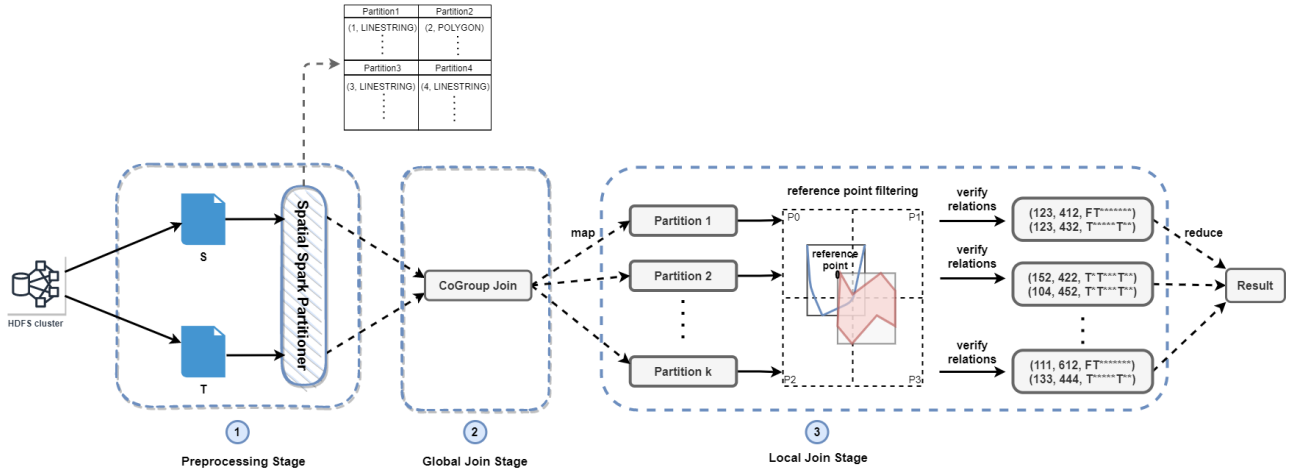

Figure 31. Spatial Spark Partitioned Join in JedAI-spatial.

4.2. Apache Sedona

Apache Sedona[20] is an open-source[21] system, immediate successor of *GeoSpark* [21], [22], and retains its key architecture, such as the Spatial RDD and Spatial Query Processing Layer. Moreover, it provides overall updates to libraries, such as JTS, as well as code improvements and bugfixes. Nevertheless, it deprecates all GeoSpark's partitioning techniques, besides Quad Trees and K-D-B-Trees and as a result the second method of duplicate elimination described below, since it is no longer applied.

JedAI-spatial provides both GeoSpark and Apache Sedona experiments. We strongly recommend using Apache Sedona, since it is more up-to-date and optimized.

In the remainder of this subsection, we first analyze GeoSpark's key features and limitations and then describe the integration of Apache Sedona in JedAI-spatial.

4.2.1. GeoSpark

GeoSpark[21] is a state-of-the-art framework that allows users to query large-scale spatial data by providing two extra layers on top of Apache Spark, by extending both the core and the Spark SQL and by supporting spatial queries in both formats. These layers are:

1. Apache Spark Layer
2. Spatial RDD (SRDD) Layer
3. Spatial Query Processing Layer

[20] https://sedona.apache.org/
[21] https://github.com/apache/incubator-sedona

The **Spatial RDD Layer** introduces a new Spatial RDD structure, written in Java, which extends Spark's RDD. SRDD supports Points, Rectangles, Polygons and LineStrings using the JTS library. Part of the SRDD Layer is the partitioning and the indexing. In more detail, GeoSpark splits the area covered by the geometries into grids using one of (i) K-D-B-Tree (ii) R-Tree (iii) Quad Tree (iv) Voronoi (v) Uniform Grid (vi) Hilbert Curves. R-Tree and Voronoi partitioning methods are implemented using sampling. As a result, the extent of the index may not cover the entire area of the geometries. For that reason, the developers of GeoSpark append the geometries that are not covered to an overflow bucket. Thereafter, the source and target geometries are mapped to grid cells and are grouped into partitions in later phases using the grid cell identifier. The framework also supports R-Tree and Quad Trees as Spatial Indices for the geometries of individual partitions.

After the SRDD has been constructed and processed, the developers may use the **Spatial Query Processing Layer** to perform Range, kNN, Spatial and Distance join queries on top of the populated SRDDs.

In order to avoid duplicate verifications, GeoSpark utilizes two techniques based on the partitioning method.

1. RPM, which we have already analyzed in various parts of the thesis, can only be applied to partitioning methods which produce disjoint partitions, such as the Quad Tree, K-D-B-Tree and Equal Grids.
2. *groupBy* operation on the result tuples, which collects all the geometries by their key and then removes the duplicates. A tuple is in the form of $(Geometry1, Geometry2)$. Using a *groupBy*, the tuples are transformed into $(Geometry1, [Geometry2, \ldots, Geometryk])$ whereby, the duplicates are removed by iterating over the array of verified geometries.

It goes without saying that the second method imposes a significant overhead since it triggers big data shuffles across the cluster.

4.2.2. Apache Sedona vs Apache Sedona in JedAI-spatial

As aforementioned, JedAI-spatial is an abstraction over the frameworks it integrates. Developers no longer need to write their own code in order to find the topological relations between their spatial data. They are required to provide the HDFS paths of their files, the partitioning method (s*edonaGridType*), the indexing method (s*edonaLocalIndexType)* and the number of partitions (*partitions*) for the cluster execution. These configurations can be found in the *configurationTemplate.yaml* file in the github repository and in ANNEX I. Apache Sedona uses the latest JTS library and has already implemented Reference Point Filtering.

Table 3. Apache Sedona vs JedAI-spatial's Apache Sedona.

	Apache Sedona	Apache Sedona in JedAI-spatial
Queries	Range, kNN, Spatial Join, Distance Join	Spatial Join

4.2.3. Process

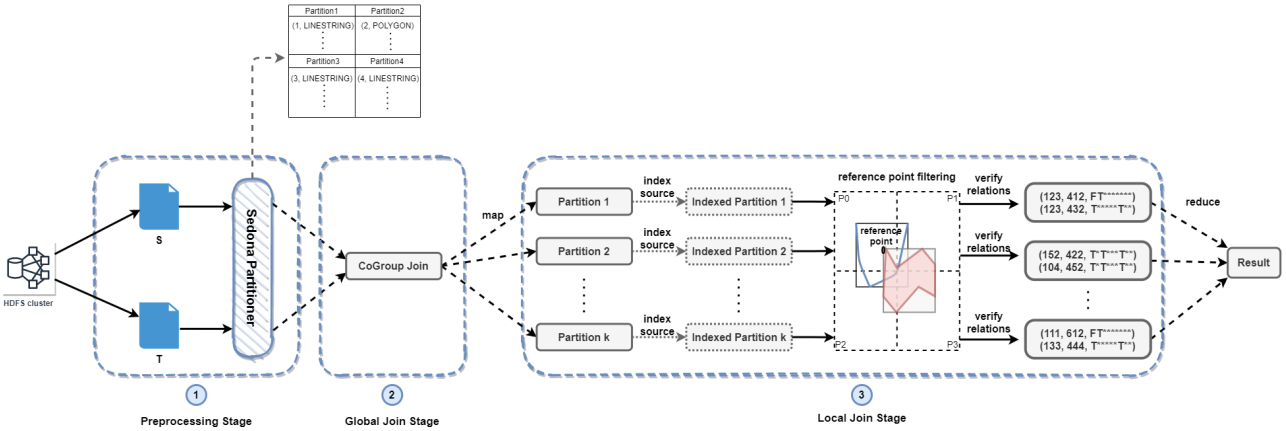

Figure 32. JedAI-spatial's Apache Sedona Process.

Figure 32 presents the actual process implemented by JedAI-spatial's Apache Sedona. Bear in mind that it is organized in three consecutive phases as indicated in the beginning of the section. In the Preprocessing Stage, the source data are partitioned based on Apache Sedona's Partitioner and then all the geometries are mapped to the corresponding partition given an identifier. SRDD partitions the data by carefully analyzing the dataset and using a user defined partition technique. In the second phase, the entities with the same identifier are grouped in the same partitions. The index source arrows in the Local Join Stage are denoted with dotted lines, because indexing the source is optional, since Apache Sedona supports both Nested Loop Index Joins and Nested Loop Joins.

4.3. Location Spark

Location Spark [23], [24] is an open-source, spatial query processing framework, available on GitHub[22], that addresses four major problems emerging in Apache Spark clusters:

1. Spatial Indexing
2. Data Skew
3. Query Optimization
4. Network shuffling

Regarding **Spatial Indexing**, Location Spark employed multiple index methods, like R-Trees, IR-Trees, EquiGrid and Quad Trees. For Global Indexing, it uses one of (i) Grid (ii) R-Tree (iii) Quad Tree to partition the input data. For Local Indexing, it uses R-Tree, IR-Trees, EquiGrid and Quad Trees.

Location Spark is the only framework that applies a form of Skew Analysis to the partitions. In Apache Spark, each partition is a processing unit. Significant overhead is imposed by not splitting the partitions evenly, due to data skew, resulting in inflated partitions and delayed execution. This deteriorates the overall performance of the cluster, because the execution cannot progress to a new job if the workers have not finished their already assigned ones. Location Spark proposes a Query Plan Scheduler, which utilized two methods to alleviate the skew problem:

- Strategy 1*:* Learns the data distribution by recording the number of data points in each partition and repartitions the data points into newly generated sub-partitions, making sure that they contain an equal amount of data.

[22] https://github.com/purduedb/LocationSpark

- Strategy 2: Collects a sample Qs from the queries Qi that are originally assigned to partitions Di and then computes how Qs is distributed over Di, by recording the frequencies of the queries in the partitions. The data are repartitioned based on the frequencies gathered.

Location Spark's **Query Optimization** chooses among a Nested Loop Join and a Nested Loop Index Join by evaluating the space and execution within each worker.

For reducing **Network Shuffling**, Location Spark employs a *Spatial Bloom Filter* which is used mostly to check if a geometry is contained in a partition. Typically, the filter is used for spatial range queries in order to find the geometries which are in range of the candidate geometry without the overhead of an unnecessary network shuffle.

4.3.1. Location Spark vs Location Spark in JedAI-spatial

JedAI-spatial has made severe changes regarding Location Spark's core. First and foremost, some vital mechanisms of Location Spark have been removed. The Query Optimization mechanism has been left up to the user. The framework no longer decides the optimal index. The index chosen is indicated by the configurations given as input by the user.

The second mechanism removed is the Spatial Bloom Filter. JedAI-spatial is a Geospatial Interlinking framework which verifies the spatial join relations. The Spatial Bloom Filter is used mainly for spatial range queries. Moreover, the geometries during preprocessing may be assigned to multiple partitions; there is no need to verify if a geometry belongs to a partition. Preliminary experiments demonstrated that Spatial Bloom Filter imposes an unnecessary overhead in our case.

Location Spark used Apache Spark's `reduceByKey` method for duplicate elimination, which is identical to `distinct`, due to triggering shuffles. We enhance the filtering phase with RPM.

The last and most important change is replacing Location Spark's partitioning phase with the GeoSpark partitioner. GeoSpark's partitioning is state-of-the-art and supports the main partitioning techniques (Quad Tree, R-Tree and EquiGrid).

However, as previously described in the Apache Sedona section, the R-Tree partitioning technique is implemented using sampling and GeoSpark[21] retains an overflow bucket for that reason. The major problem for JedAI-spatial is that the overflow bucket contains geometries from the entire input datasets, meaning it cannot leverage the reference point method to remove the duplicates, because the extent MBR of those geometries is not disjoint with all the MBRs of the other partitions, a problem described in GeoSpark paper as well. Given that we want to avoid using the second method for duplicate elimination of GeoSpark, due to its high computational cost, we completely removed R-Tree from the available partitioning techniques.

In the future, JedAI-spatial could replace R-Tree partitioned with *R-Grove* [35], which offers disjoint partitions and cares for the problems addressed below.

Table 4. summarizes all the changes between Location Spark and JedAI-spatial's Location Spark.

	Location Spark	Location Spark in JedAI-spatial
Skew Analysis	Strategy 2	Strategy 1
Data Types	Point, Rectangle	Rectangle, Polygon, LineString
Queries	Range, kNN, Spatial Join, Distance Join, kNN Join	Spatial Join
Duplicate Elimination	`reduceByKey`	reference point technique
Partitioning Techniques	Grid, Quad Tree, R-Tree	GeoSpark Partitioner
Local Index	R-Tree, IR-Tree, EquiGrid, Quad Tree	R-Tree, EquiGrid, Quad Tree

4.3.2. Process

JedAI-spatial's Location Spark's processing is depicted in Figure 33. Initially, it divides the input data through GeoSpark's Partitioner. Then, Location Spark's Query Plan Scheduler performs a skew analysis in order to partition the data as evenly as possible; it uses Strategy 2, which repartitions the skewed partitions which are at least twice the size of the smallest partition. After inner joining partitions with the same id, a local index is constructed for each partitioning (R-Tree, QuadTee, EqualGrids) and the topological relations are verified based on DE-9IM.

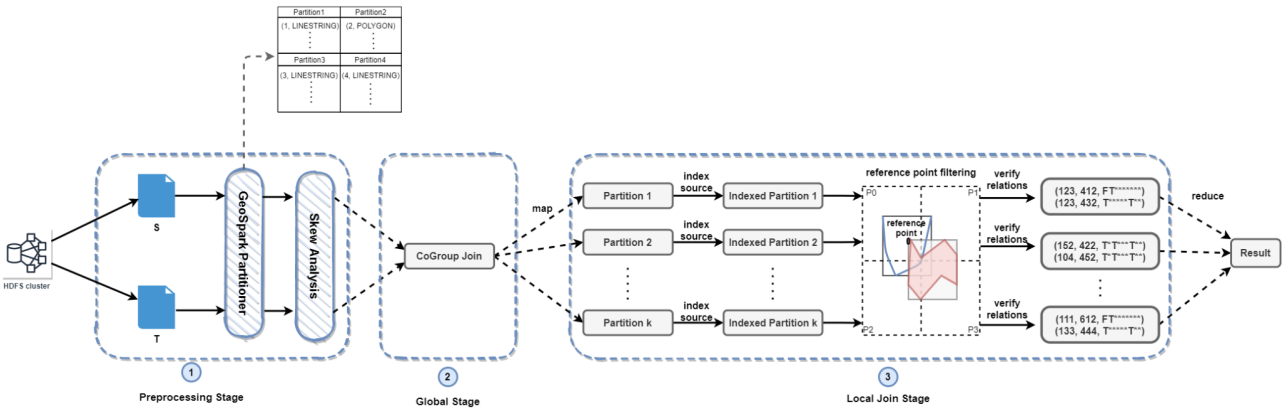

Figure 33. JedAI-spatial's Location Spark Process.

4.4. Magellan

Magellan[23] is the fourth framework integrated in JedAI-spatial. It must be noted that Magellan is solely an open-source GitHub project that extends the Spark SQL module of Apache Spark with extra rules. Nevertheless, it is referenced in multiple experimental evaluation sections in the literature. It is based on Z-Order/Morton Curves [25]. Z-Order Curves are space filling curves which Morton introduced as a system in order to sequence a static two-dimensional geographical database. This system allowed the division of the Earth's surface from two-dimensions to one-dimension preserving *spatial locality*, meaning the geographical close locations are portrayed close in the coordinate system and consequently to non-volatile media.

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

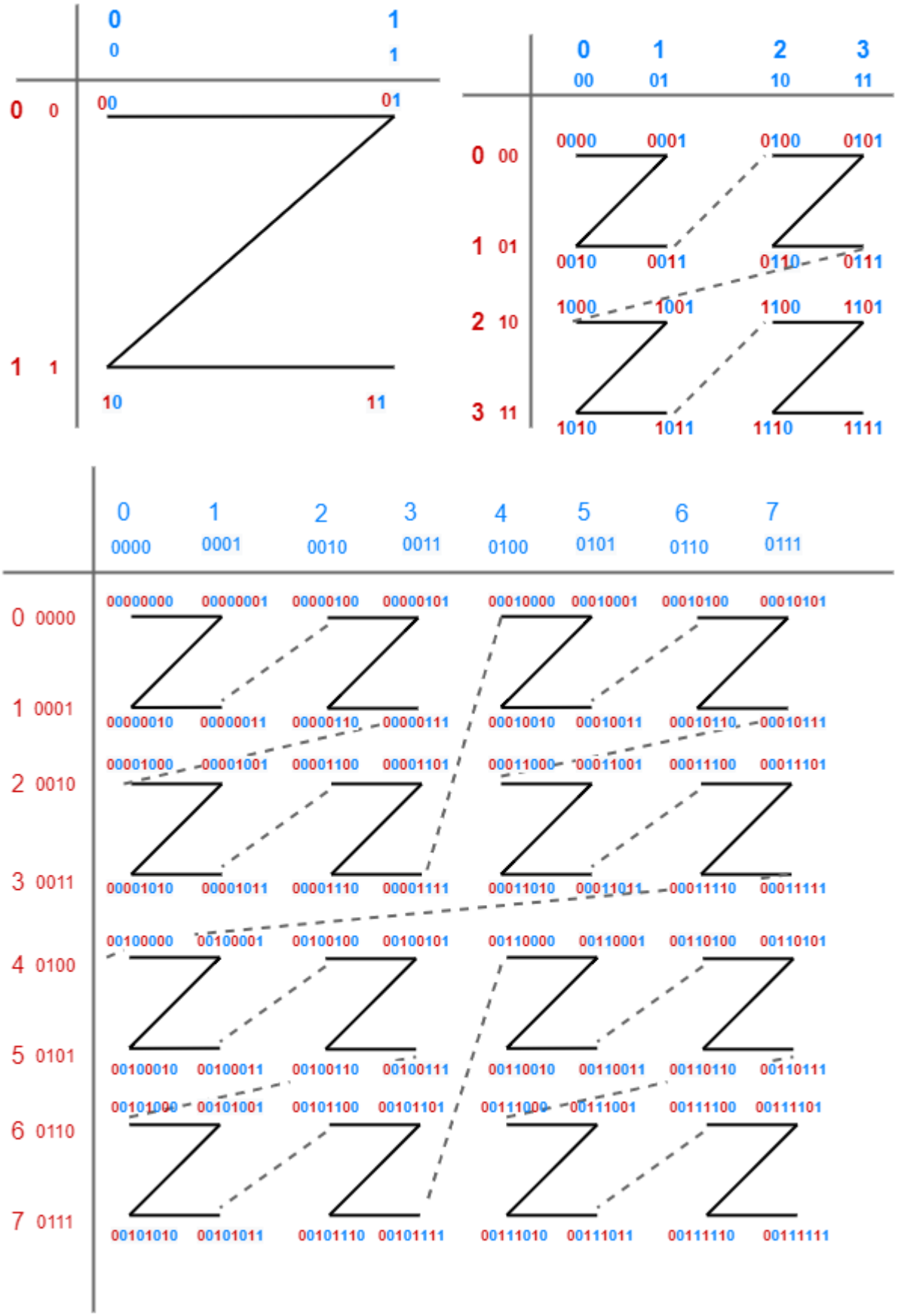

Figure 34. Z-Order Curve Overview.

In Morton's proposition, the space is divided into *frames/tiles*. Each tile is represented by a pair of (x, y). The value of a tile is calculated by interleaving the binary representations of (x, y). Connecting the value of each tile in a sequence produces the Z-Order curve. The way the coordinates are sequenced resembles a Z, hence the name.

Example 7. Figure 34, consists of 3 subfigures. In the top left, the Z-Order curve has a precision of 2 and the space it fills is partitioned in 4 tiles. These 4 unit frames make a 1-factor frame. In the top right, the Z-Order curve has a precision of 4 and it divides the space in 16 tiles. These 4 1-factor frames combined, produce a 2-factor frame. At last, the final figure consists of a Z-Order curve with a precision of 6 dividing the space in 64 tiles, producing a 3-factor frame, meaning 4 2-factor frames. Each f-factor frame is created by using lower f-factor frames.

Example 8: *Find the tile of coordinates* $(4, 5)$. Firstly, find the binary representation of 4, which is 100 and 5, which is 101. The interleaving representation of $(4, 5)$ is 110010, hence 50.

4.4.1. Magellan vs Magellan in JedAI-Spatial

Following are the differences between Magellan and JedAI-spatial's Magellan. The most notable improvement is the transformation from the Dataframe API to Spark RDDs. Magellan does not include any sophisticated method for duplicate elimination. The Dataframe API supports two methods; `distinct` and `dropDuplicates`, which are both computationally expensive. We have replaced them with RPM.

Table 5. Magellan vs JedAI-spatial's Magellan.

	Magellan	Magellan in JedAI-spatial
Language	Extended Spark SQL	Spark RDD
Data Types	Point, LineString, Polygon, MultiPoint, MultiPolygon	Rectangle, Polygon, LineString
Queries	Range, Spatial Join	Spatial Join
Duplicate Elimination	`distinct` / `dropDuplicates`	reference point filtering

4.4.2. Process

Magellan leverages Z-Order curves to partition the space. Unlike Spatial Spark Partitioned Join, which considers the extent of the source and target geometries, Magellan considers the extent of Earth's surface as an MBR of (-180, 180, -90, 90). It allows users to associate every geometry with one or more Z-Order Curves. Original Magellan gave users the choice to not use a Z-Order Curve Index at all, resulting in a cartesian join between the source and target dataset. JedAI-spatial has completely deprecated this approach, because it does not scale to large datasets, due to memory limitations, and always uses Z-Order Curves.

In the Preprocessing Phase, the initial $RDD[Geometry]$ is transformed into $RDD[Geometry, Array[Z-OrderCurve]]$. These curves cover the bounding box of the respective geometry and are used to join the spatial objects into the same partitions using the interleaving bits of their (x, y) coordinates. This technique acts as a partitioning scheme, dividing the space into $2^{precision}$ tiles. In due course, each partition is assigned to

an executor, where a nested loop join is performed and the final relations are produced. The whole process is illustrated in Figure 35.

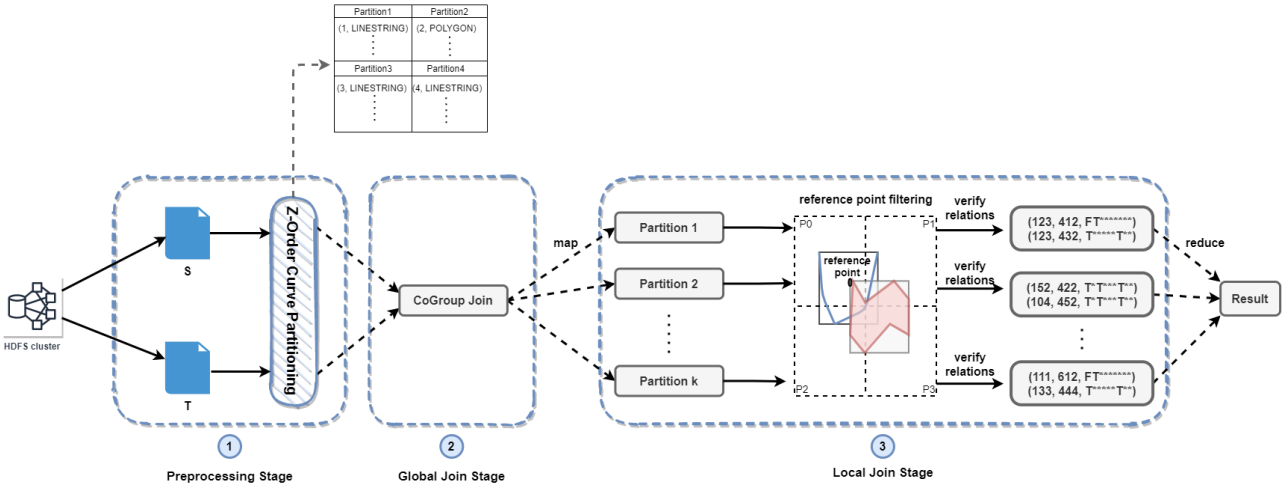

Figure 35. JedAI-spatial's Magellan Process.

5. EXPERIMENTAL RESULTS

In this section, we present the experimental evaluation of each spatial join algorithm carried out on a server with Intel Xeon E5-4603 v2 @ 2.20GHz, 128GB RAM, consisting of 32 cores and 4 NUMA nodes. The parallel methods are implemented using Scala 2.11.12 and Apache Spark 2.4.4.

The datasets used for the experiments are popular in the literature [26], [27] and comprise real datasets imported from US Census Bureau TIGER[24] files. They consist of USA's Area Hydrography (AREAWATER), Linear Hydrography (LINEARWATER), roads (ROADS) and edges (EDGES) as well as datasets extracted from OpenStreeMap representing lakes (Lakes), parks (Parks), roads (Roads) and buildings (Buildings) of the whole world. The datasets are combined into six pairs (D_1-D_6), as illustrated on Table 6:

Table 6. Technical characteristics of the real datasets used in our experimental analysis.

	D_1	D_2	D_3	D_4	D_5	D_6
Source Dataset	AREAWATER	AREAWATER	Lakes	Parks	ROADS	Roads
Target Dataset	LINEARWATER	ROADS	Parks	Roads	EDGES	Buildings
#Source Geometries	2,292,766	2,292,766	8,419,320	9,961,891	19,592,688	72,339,926
#Target Geometries	5,838,339	19,592,688	9,961,891	72,339,926	70,380,191	114,796,567
Cartesian Product	1.34×10^{13}	4.49×10^{13}	8.19×10^{13}	7.11×10^{14}	1.38×10^{15}	8.30×10^{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	5,551,014	14,163,325	163,982,135	1,041,562
#Contains	806,158	3,792	947,788	6,323,433	12,218,867	276,010
#CoveredBy	0	0	3,031,403	48,922	53,758,452	83,936
#Covers	832,843	4,692	948,086	6,470,655	12,218,867	276,023
#Crosses	40,489	106,823	270,248	6,490,937	6,769	314,708
#Equals	0	0	557,465	3,147	12,218,867	18,972
#Intersects	2,401,396	199,122	5,551,014	14,163,325	163,982,135	1,041,562
#Overlaps	0	0	822,241	45,116	73	54,899
#Touches	1,554,749	88,507	1,037,412	1,258,163	110,216,841	332,249
#Within	0	0	3,030,790	48,823	53,758,452	82,668
Total Topological Relations	5,635,635	402,936	16,196,447	34,852,521	418,379,323	2,481,027

Following is a figure illustrating the size of the cartesian product of each dataset, which provides a measure for the volume of data we considered.

[24] http://spatialhadoop.cs.umn.edu/datasets.html

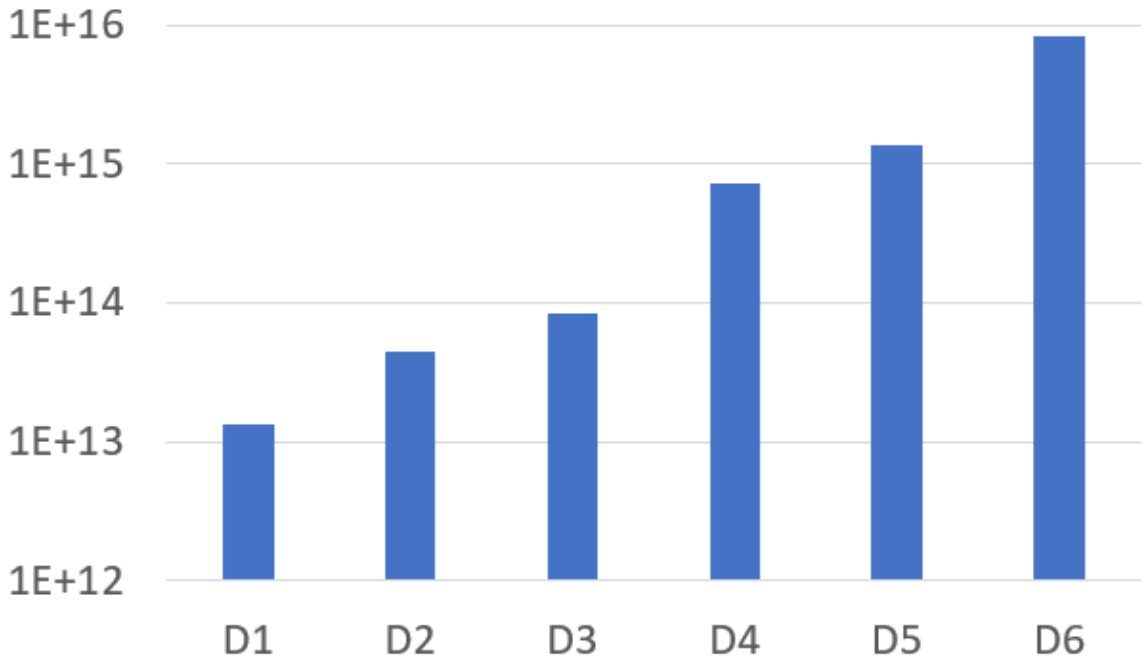

Figure 36. Cartesian product of geometry pairs in D1-D6.

To show the robustness of each method as well as their scalability as the size of the input data increases, D_1 was divided into 10 distinct subsets of increasing size. Each subset contains a percentage of the total relations of D_1 and is bigger than its previous by 10%. Special care was taken to ensure that every subset captures correctly the corresponding computational cost both with respect to the number of geometries from each input dataset and the number of related pairs. The sample datasets based on D_1 are available at Table 7:

Table 7. D1's and subsets' topological relations.

	10%	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	***D*$_1$**
#Source Geometries	229,276	458,553	687,829	917,106	1,146,383	1,375,659	1,604,936	1,834,212	2,063,489	2,292,766
#Target Geometries	583,833	1,167,667	1,751,501	2,335,335	2,919,169	3,503,003	4,086,837	4,670,671	5,254,505	5,838,339
#Contains	103,260	213,447	324,852	432,286	532,688	623,290	699,244	757,131	793,947	806,158
#CoveredBy	0	0	0	0	0	0	0	0	0	0
#Covers	108,976	224,501	340,698	451,795	554,831	647,392	724,686	783,328	820,525	832,843
#Crosses	6,446	13,266	19,096	24,582	29,095	32,546	35,564	37,769	39,375	40,489
#Equals	0	0	0	0	0	0	0	0	0	0
#Intersects	312,327	656,523	1,000,117	1,329,225	1,628,178	1,887,767	2,102,013	2,261,258	2,362,497	2,401,396
#Overlaps	0	0	0	0	0	0	0	0	0	0
#Touches	202,621	429,810	656,169	872,357	1,066,395	1,231,931	1,367,205	1,466,358	1,529,175	1,554,749
#Within	0	0	0	0	0	0	0	0	0	0
Total Topological Relations	733,630	1,537,547	2,340,932	3,110,245	3,811,187	4,422,926	4,928,712	5,305,844	5,545,519	5,635,635

5.1. Serialized Experiments

All serialized methods described above produce the same results. Therefore, we can only assess their relative performance with respect to their time efficiency: the lower the run-time of a method is, the better is its performance. To this end, we carried out two experiments:

1. We measure their scalability over the increasing input size of the 10 subsets of D_1
2. We measure their run-time over datasets D_1 - D_3.

In each case, we are interested in two efficiency measures, with each one which estimating the time required to complete the homonymous step in the Filter-Verification framework:

1. The filtering time, t_f.
2. The verification time, t_v.

In this way, we are able to examine the impact of every step on the overall performance of each method and to provide insights into the strengths and weaknesses in each case.

Due to the large number of methods implemented by JedAI-spatial, we present their performance in three groups: (i) the grid-based methods, which include GIA.nt and RADON along with their static variants, (ii) the partition-based methods, which include Plane Sweep and its variants, and (iii) the tree-based methods, which include R-Tree, Quad Tree and CR-Tree. We applied every method to every dataset 10 times and considered the average run-times. Below, we describe the results of every experiment.

5.1.1. Scalability analysis

The resulting filtering and verification times appear in Figure 37 and Figure 38, respectively. In each figure, the same scale is used in all diagrams, to facilitate the comparisons between the three categories.

Starting with Figure 37, we observe that for all algorithms, the Filtering step is completed within a few seconds, 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 manages to reduce the number of candidates by a whole order of magnitude, as indicated by the relative size of the Cartesian Product and the number of Candidate Pairs (with intersecting MBRs) in Table 6.

Among the grid-based methods, RADON is consistently the slowest approach, followed by GIA.nt, whose run-time is lower by at least 30%. The reason is that RADON iterates over both input datasets, whereas GIA.nt considers exclusively the source one. In both cases, though, their static variants are significantly faster: Static RADON is faster than its dynamic counterpart by ~40%, while Static GIA.nt outperforms GIA.nt by 60%-75%; in the latter case, the larger the subset of D_1, the larger is the difference between the two methods, because Static GIA.nt increases its t_f by just 3 times from 10% to 100%, unlike GIA.nt, which increases its t_f by 12 times (i.e., linearly).

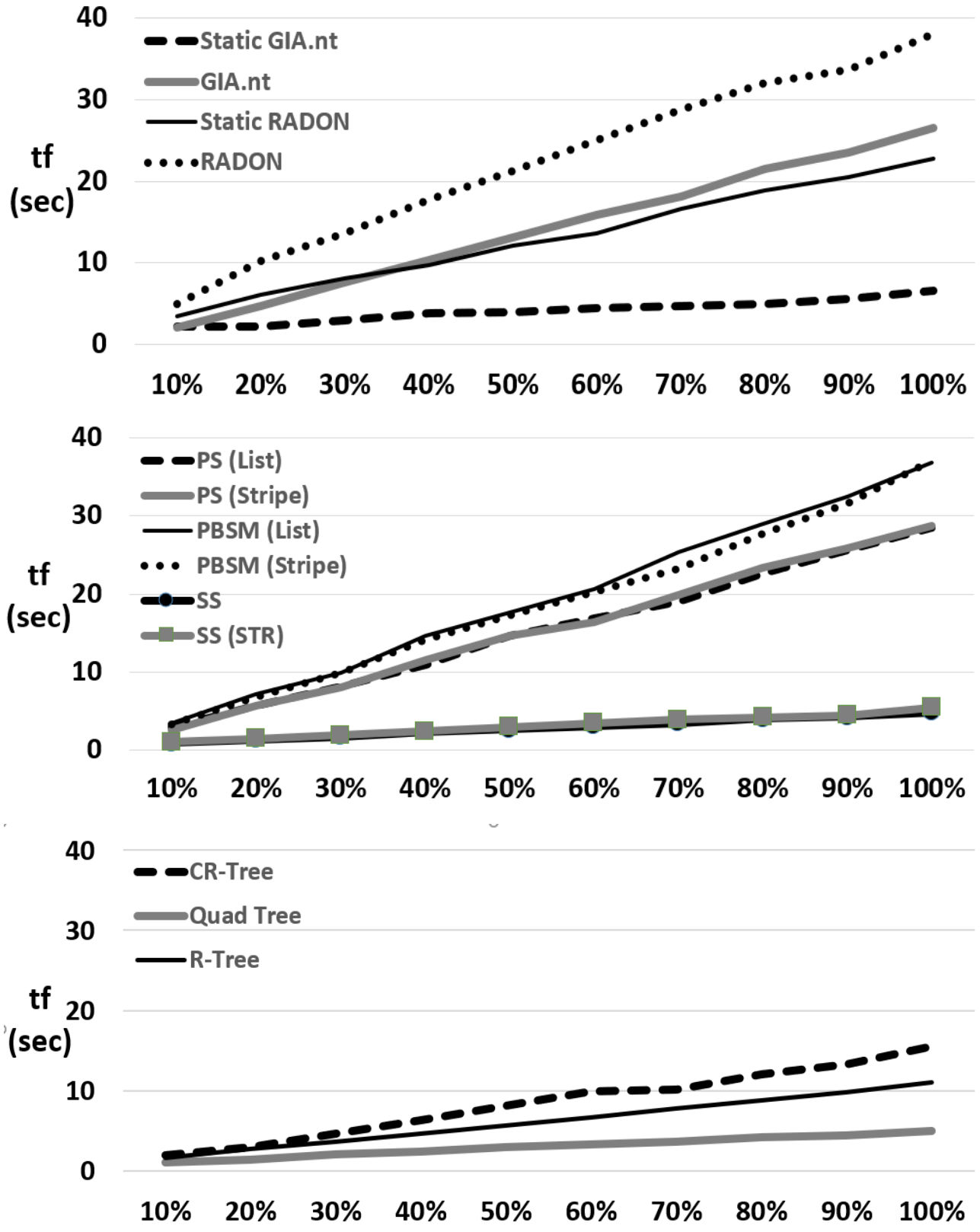

Figure 37. Filtering time of each serial algorithm over the scalability datasets of Table 7.

Regarding the partition-based methods, we observe that for each main algorithm, the filtering time is almost identical for the two variants we consider, regardless of the underlying data structure. We also observe that Strip Sweep involves the fastest, by far, filtering step. This should be attributed to its simplicity, since it indexes only the source geometries in a limited number of strips. As a result, it scales sublinearly with the increase of the input data: from 10% to 100%, its t_f raises by just 5 times. Strip Sweep is followed by Plane Sweep, which is 4-5 times slower, with PBSM being 15%-20% slower. This should be expected, given that PBSM applies Plane Sweep to each one of its space partitions. Both algorithms, though, scale linearly with the increase in the input data.

Finally, all tree-based algorithms exhibit a sublinear scalability, as they index exclusively the source dataset. Quad Tree is the fastest algorithm, followed by R-Tree and CR-Tree. The last algorithm involves a significant overhead for the compression of the resulting tree index. Note that Quad Tree involves the overall fastest Filtering, together with Stripe Sweep.

Regarding the Verification time, we notice in Figure 38 that it constitutes the bottleneck of Geospatial Interlinking, being two orders of magnitude larger than Filtering time. This is caused by the complexity of the input geometries, which determines the time complexity for calculating a single Intersection Matrix.

Looking into the grid-based algorithms, we observe that (Static) RADON is significantly faster than (Static) GIA.nt - from 5% to 12%. The larger the input datasets are, the larger is their difference. This is in contrast with the relative performance reported in [18], where GIA.nt is slightly faster than RADON, demonstrating the significant impact of the implementation improvements we have incorporated in JedAI-spatial. Note, though, that the lower run-time of (Static) GIA.nt should be attributed to the overhead of reading the target geometries from the disk, unlike (Static) RADON, which loads the target dataset into main memory during the Preprocessing phase; this overhead increases linearly with the size of the input data, thus accounting for the larger differences between the two algorithms as we move from 10% to 100%. It is also worth noting that Static GIA.nt is slightly slower than GIA.nt, because of the finer-grained tiles it employs, which increase the tiles associated with every geometry and raise the overhead of retrieving the candidates per target geometries.

Among the partition-based algorithms, we notice that Plane Sweep and PBSM are by far the slowest ones when using Linked Lists to maintain the active source and target geometries. This is caused by the high overhead of querying and updating the contents of the Linked Lists. The situation is actually worse for PBSM, which maintains separate lists for each partition. The performance of both algorithms is significantly improved when using Stripes, which significantly reduces the overhead of maintaining the active geometries. Plane Sweep with Stripes is actually the fastest partition-based algorithm, followed in close distance by PBSM, which is 3% to 7% slower, due to the overhead of RPM, which is applied to every pair before its verification. Almost identical performance is exhibited by Stripe Sweep with STR trees, which significantly improves the plain Stripe Sweep algorithm: by considering the overlap of candidate pairs on the vertical axis, it reduces the verification time from 11% over the smallest dataset to 29% over the largest ones, because the number of irrelevant candidates in stripes increases with the increase of target geometries. Note though that Stripe Sweep reads the target geometries from the disk, one by one, unlike Plane Sweep and PBSM, which load them into main memory, during the Preprocessing phase.

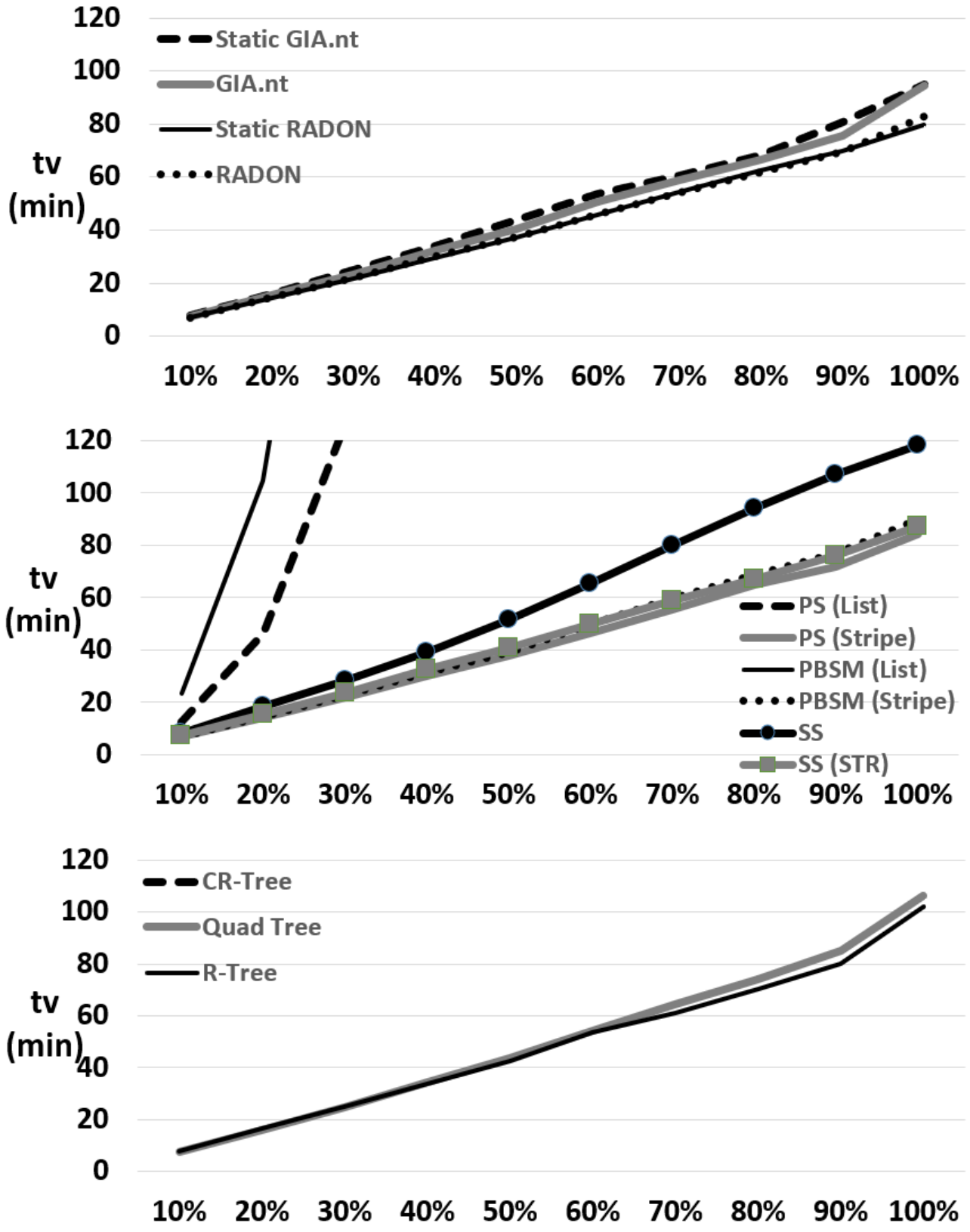

Figure 38. Verification time of each serial algorithm over the scalability datasets of Table 7.

Finally, we observe that Figure 38 reports only the performance of Quad Tree and R-Tree. CR-Tree is excluded, because its run-time over the smallest dataset is 235 minutes, exceeding the time required by most other algorithms even for the largest dataset. The reasons are (i) its high cost of retrieving the candidates for every target geometry, due to the required transformation of the MBRs, and (ii) the lack of memory manipulation Java provides compared to languages like C or C++. Such low-level languages allow the user to allocate amounts of memory on demand and as a result create structs of size that can be leveraged by the L1 and L2 cache. Quad Tree and R-Trees process every query in an efficient way, thus yielding much lower verification times. The latter is actually slightly faster than the former, due to its lower overhead. Recall that both algorithms read the target geometries from the disk on the fly.

Overall, all serialized algorithms involve a rather efficient Filtering step, which reduces the search space by orders of magnitude. Yet, their performance is determined by the efficiency of the Verification step, which is the bottleneck of Geospatial Interlinking. In this respect, the fastest algorithms are RADON and Static RADON, followed in close distance by Plane Sweep based on Stripes. Yet, these algorithms are **memory-intensive**, requiring that the target geometries can be loaded into main memory. As a result, they cannot scale to large datasets, as discussed below. Among the **memory-frugal** algorithms, which read the target geometries one by one from the disk, GIA.nt involves the fastest verification, with Stripe Sweep being slightly slower.

5.1.2. Performance over D_1 - D_3

In the previous sub-section, we compared thoroughly the pros and cons of each algorithm. We now present the performance of the serial processing algorithms for datasets D_1 - D_3 in Table 8 and Table 9 in seconds and hours respectively. Figure 39 and Figure 40 illustrate the filtering and verification time respectively.

Table 8. Filtering time of serial processing algorithms for datasets D_1-D_3 in seconds.

	D_1	D_2	D_3
Static GIAnt	6.85	9.84	75.02
GIA.nt	34.82	32.10	117.97
Static RADON	24.93	35.81	
RADON	38.99	88.71	
Plane Sweep (Stripe)	33.42	72.35	
PBSM (Stripe)	38.09	97.29	
Strip Sweep	5.56	5.01	20.54
Strip Sweep (STR)	5.24	5.97	25.50
QuadTree	6.83	4.70	28.97
R-Tree	13.03	12.95	53.01

Table 9. Verification time of serial processing algorithms for datasets D_1- D_3 in hours.

	D_1	D_2	D_3
Static GIAnt	1.58	3.35	9.54
GIA.nt	1.58	3.42	9.58
Static RADON	1.33	2.94	
RADON	1.38	2.91	
Plane Sweep (Stripe)	1.40	2.96	
PBSM (Stripe)	1.50	3.36	
Strip Sweep	1.97	4.55	13.90
Strip Sweep (STR)	1.45	3.09	9.68
QuadTree	1.77	3.48	11.10
R-Tree	1.70	3.36	56.99

We do not take into account datasets D_4 - D_6, because of their long-lasting execution time. Note also that the memory-intensive algorithms are excluded from the evaluation of dataset D_3 due to insufficient memory. Recall that such algorithms load both the source and target datasets in main memory, but this is not possible with the available 128GB of RAM. These are (Static) RADON, Plane Sweep and PBSM. The rest of the algorithms are memory-frugal, building spatial indexes only on the source dataset.

Starting with the grid-based algorithms, we observe that the static variants, Static RADON and Static GIA.nt have a faster filtering time compared to the non-static, RADON and GIA.nt. This observation occurred in the scalability analysis as well, because the static variants do not find the average width of every geometry. The verification time of static variants is identical with the non-static variants.

Regarding the partition-based algorithms, PBSM has by far the longest filtering-time, due to its coarse-grained tiles (i.e., it sorts a large number of geometries per tile). Strip Sweep has the most simple filtering phase resulting in the best filtering time among the group. Strip STR Sweep seems to have the same filtering time as Strip Sweep, due to the fact that the STR-Tree is constructed when the first `query` method is called. Hence, a significant part of the filtering is within the verification.

As it is apparent, the tree-based methods' filtering time is significantly lower than the other algorithms. R-Tree is on par with Quad Tree for datasets D_1 and D_2. For dataset D_3, its verification time is the highest among all the algorithms (almost 60 hours). We believe this happens because of poor configuration (M=6) and/or because D_3 has a lot of overlapping MBRs. As a result, when range searching for candidate geometries, the subtrees visited recursively deviate from the average complexity of $O(log_M(n))$. This is a well-known problem for the R-Trees. Using an R-Tree with STR tree packing algorithm could alleviate such problems as already mentioned.

To conclude, the fastest methods are Quad Tree, GIA.nt and Strip STR Sweep. Their key characteristic is their simplified and robust filtering phase method, which scales sub-linearly among different datasets. Their efficient index also allows for faster

verification and faster execution, in total, unlike methods like R-Tree, whose verification time is heavily affected by overlapping partitions in the spatial index. Finally, the memory-intensive algorithms are only suitable for datasets small enough to fit into main memory. In these cases, they provided competitive run-times, especially for the Verification phase, as they save the cost of loading the target dataset from the disk on-the-fly.

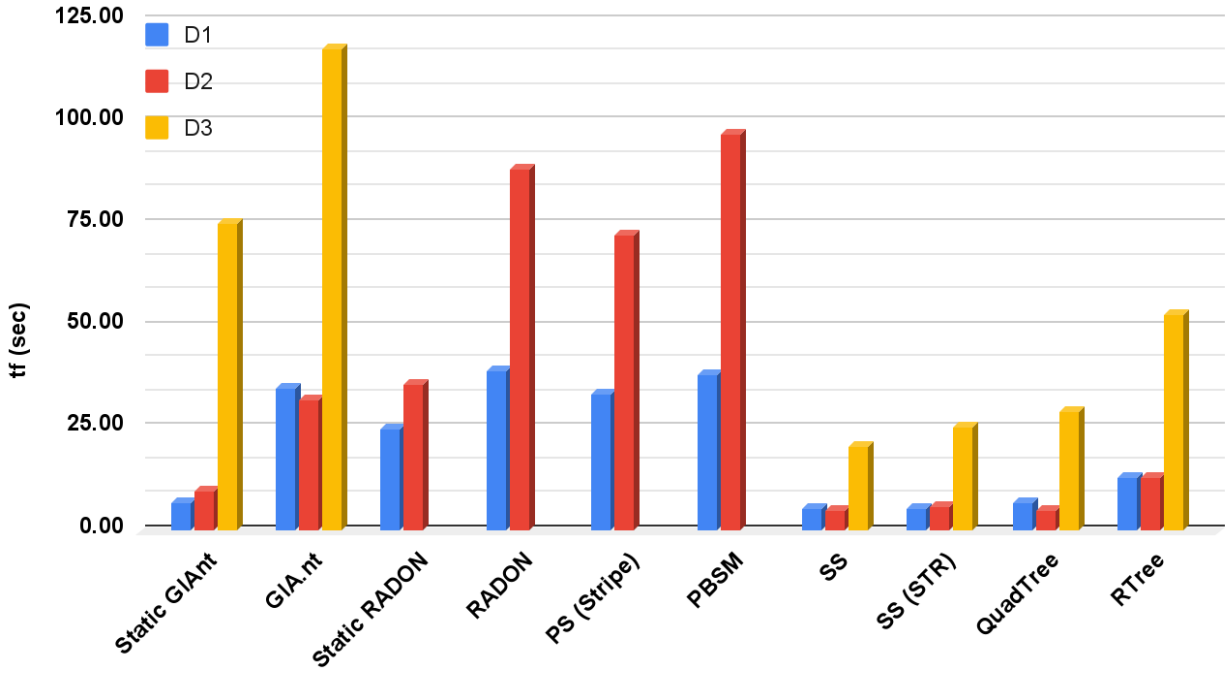

Figure 39. Filtering time of each serial algorithm over D_1-D_3.

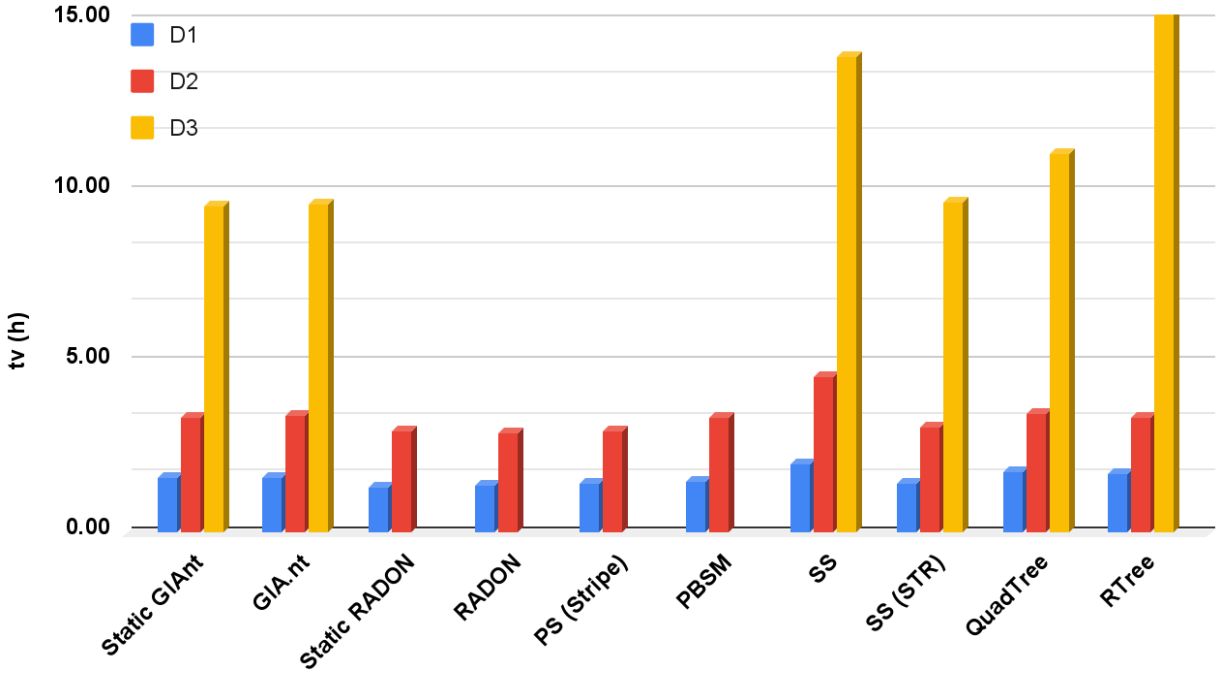

Figure 40. Verification time of each serial algorithm over D_1-D_3.

5.2. Parallel Experiments

From the parallel experiments, we exclude Spatial Spark with broadcast join due to the KryoSerializer limitation analyzed earlier in the thesis. Moreover, it should be stressed that we do not present the execution time for each phase (preprocessing, global join, local join) due to Apache Spark's internal functionality: Apache Spark has two kinds of methods, *transformations* and *actions*. Transformations are always lazy, meaning they do not make any calculations until an action is invoked by the driver program. Adding actions before and after each phase so as to measure them, is not a good practice because Apache Spark's job scheduling and DAG scheduler pipeline many transformations together for optimization. As a result, applying the above technique would produce inaccurate results. For that reason, we exclusively report the total execution time for each method.

We conducted three major experiments so as to examine:

1. The scalability of each algorithm over the subsets of D_1 in Table 7. (Section 5.2.1)
2. The vast gap in performance between reference point method and Apache Spark's `distinct` method for eliminating the redundant candidate pairs in each method. (Section 5.2.2)
3. An evaluation between the best configurations for each method on the six real large datasets of Table 6. (Section 5.2.3)

5.2.1. Scalability analysis

We conducted numerous experiments with each method in order to find the optimal configurations. In more detail, we consider the following parameters:

- We combined Spatial Spark with Fixed Grid and Sort Tile Partitioning using 32x32, 64x64, 128x128, 256x256, 512x512 and 1024x1024 grids. The sampling ratio for Sort Tile Partitioning was set to 10% of the overall source dataset. However, more effort is required in order to come up with a better method for sampling, due to the fact that most cases were error-prone regarding the verifications. The bigger the dataset, the more incorrect were the results. We also experimented with the Binary Split Partitioning, but its performance was below expectations. In a later version of JedAI-spatial, a more optimized implementation will take its place.
- For Magellan, we tested the values 16, 20 and 24 for the Z-Order Curve precision. Magellan seemed to have a huge disk spill for a Z-Order Curve precision of 24 when executing on 20% or more of D_1.
- For Apache Sedona, we used both K-D-B-Tree and Quad Tree partitioning with R-Tree and Quad Tree local indexing. We also performed experiments without local indexes at all (Nested Loop Join).
- We combined Location Spark with Quad Tree partitioning and Quad Tree, R-Tree and EqualGrid local indexing.

The experiments ran over the subsets of D_1 in Table 7 are 240 in total. In Table 10, we present the frameworks with the best configurations along with their total wall-clock execution time in seconds. These are Apache Sedona with K-D-B-Tree partitioning and R-Tree local indexing (AS KDB-RT), Spatial Spark with 512x512 Fixed Grid Partitioning (SP FGP), Location Spark with Quad-Tree partitioning and R-Tree local indexing (LS QT-RT) and Magellan with Z-Order Curves of 20 precision.

Table 10. Total execution time for D_1.

	10%	20%	30%	40%	50%	60%	70%	80%	90%	D_1
AS KDB-RT	165.20	223.20	231.61	309.30	347.19	431.03	550.85	614.66	661.05	843.67
SP FGP	211.87	284.03	383.08	512.13	640.69	748.69	922.97	1189.47	1360.44	1472.87
LS QT-RT	214.10	236.10	275.68	405.65	485.38	372.92	490.84	534.57	583.55	664.73
Magellan (20)	191.74	259.00	331.65	410.67	488.22	577.45	686.99	794.81	902.56	1037.94

Each algorithm is successful for different amounts and types of data. As it is apparent, the algorithms that implement a Local Index scale better to big data. This means that implementing a Local Index trades higher space requirements for better run-times.

Spatial Spark with Fixed Grid Partitioning is performing poorly for all the subsets of D_1. This happens due to the fact that using a Fixed Grid for partitioning does not partition the geometries effectively: in cases of fine-grained tiles, it assigns every geometry to multiple partitions, thus increasing the filtering computations. However, this was the best performance we achieved with Spatial Spark. Location Spark is also performing poorly for small datasets until its skew analysis algorithm bears fruit starting from the 60% of D_1. This is depicted with a slight curve from 50% to 60% in Figure 41. Magellan performs approximately as the average between Spatial Spark and Apache Sedona, which is the top performer for the smallest subsets, due to its effective partitioning during the Global Join Phase.

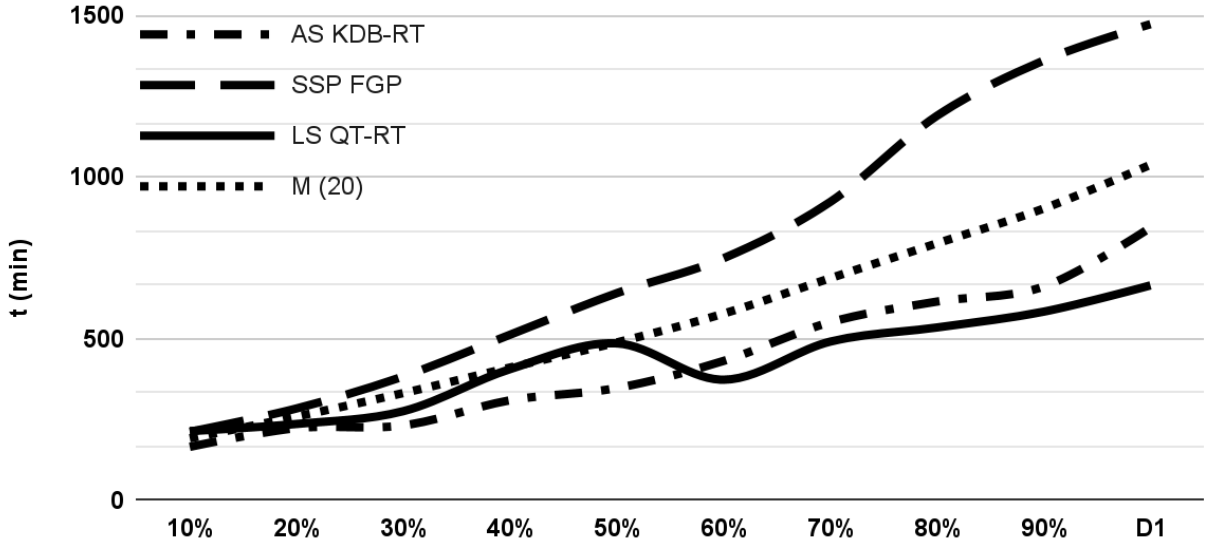

Figure 41. Wall-clock execution run-time of each parallel algorithm over the scalability datasets of Table 7.

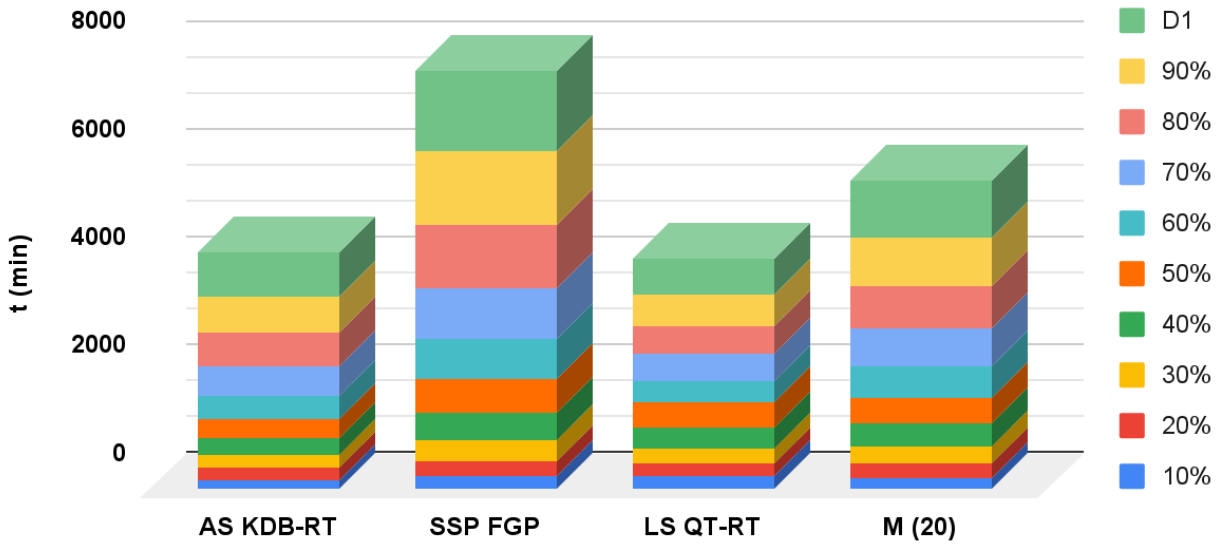

Figure 42. Stacked column chart for total execution time of D1.

It is also worth stressing that all algorithms scale sublinearly with the size of the input data: their wall-clock run-time increases by 3 times (LocationSpark) to 7 times (Spatial Spark) when comparing 10% with 100%. Moreover, when comparing their run-time with that of serialized algorithms in Figure 37 and Figure 38, we notice that they are significantly faster, especially for the larger subsets, where the overhead of Apache Spark pays off: for the entire D_1, the slowest parallel algorithm (Spatial Spark) is 3.2 times faster than the faster serialized algorithm (RADON).

Considering their overall wall-clock execution over all subsets in Table 10, we observe in Figure 41 that Location Spark is the most efficient approach, followed in close distance by Apache Sedona. Both algorithms require almost half the overall run-time of Spatial Spark, with Magellan lying in the middle of these two extremes.

5.2.2. Reference Point Method vs Spark.distinct()

To show the strength of the reference point method, we run an experiment on the first 10% data of D_1 using Spatial Spark Partitioned Join with 256x256 Fixed Grid Partitioning.

The experiment utilizing RPM, lasted 199.63 seconds. The same experiment using Apache Spark's `distinct` method lasted 40 minutes, whereafter the execution stopped because the overall shuffle writes surpassed the available memory and disk. The shuffle writes reached 130.4 GBs before the process terminated. The larger the volume of data the more the overall size of the shuffles and subsequently the execution time.

Following are screenshots from Apache Spark's UI denoting the execution time and shuffle writes:

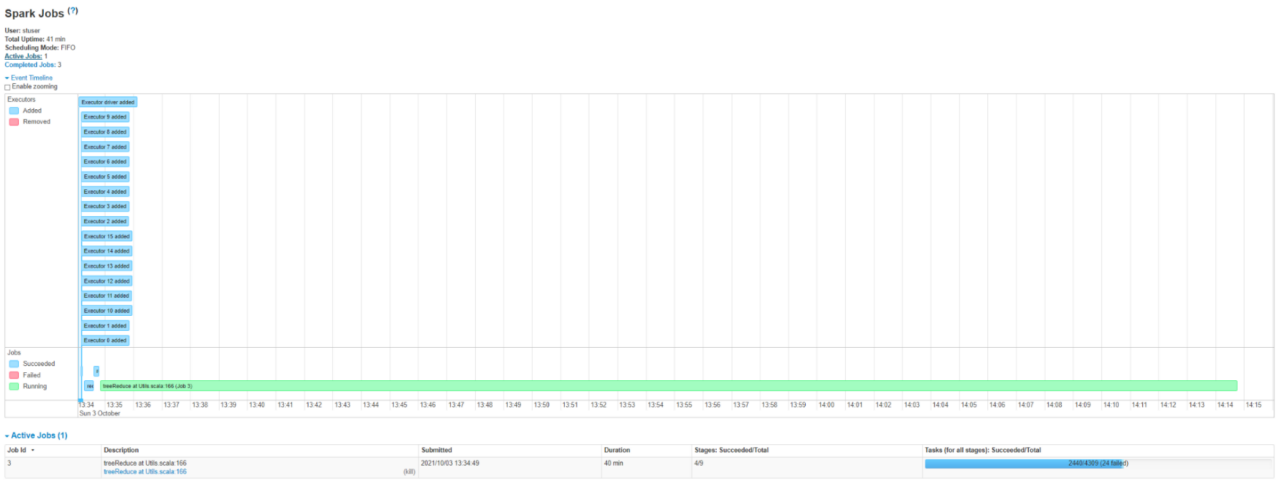

Figure 43. Apache Spark UI execution time for Spatial Spark with distinct filtering.

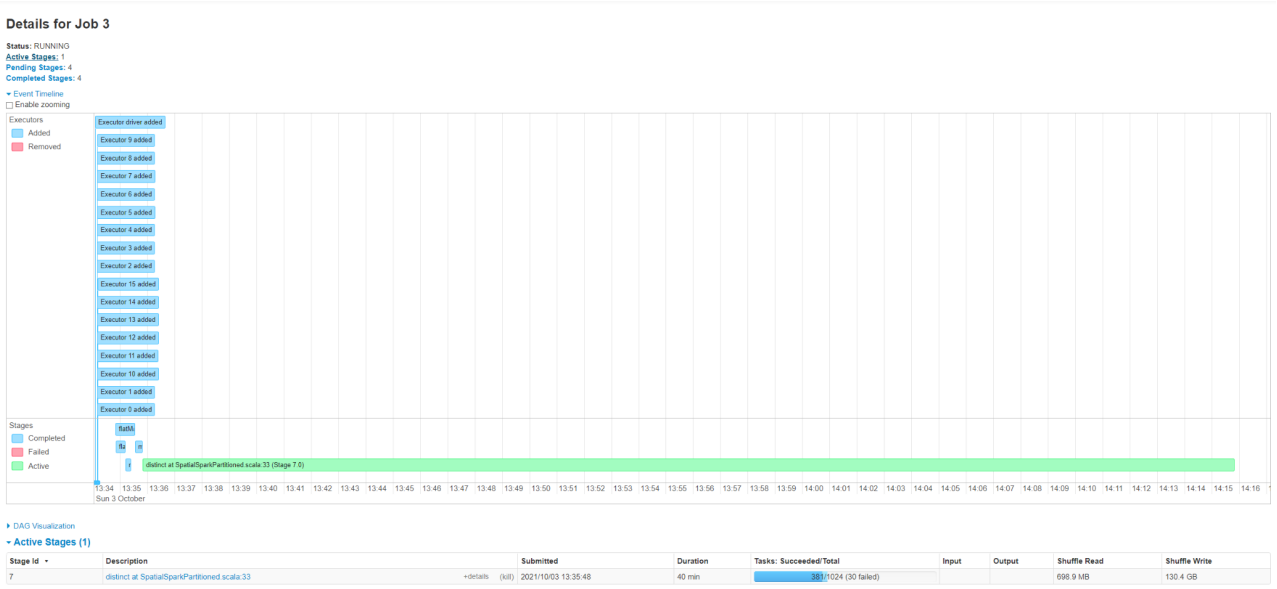

Figure 44. Apache Spark UI shuffle write for Spatial Spark with distinct filtering.

5.2.3. Performance over D_1 - D_6

For this subsection, we executed 84 experiments based on the best performing frameworks from subsection 5.2.1. These were: Spatial Spark with Fixed Grid (512x512, 1024x1024) and Sort Tile (128x128, 256x256, 512x512, 1024x1024) partitioning with 10% sampling ratio, Magellan (20 Z-Order Curve precision), Apache Sedona with Quad Tree or K-D-B-Tree partitioning and R-Tree or Quad Tree local indexes and lastly, Location Spark with Quad Tree partitioning and Quad Tree, R-Tree or EqualGrid local indexing.

From the above-mentioned configurations, we present the frameworks with the best ones. Table 11 and Figure 45 report the total wall-clock execution time for datasets D_1 - D_6 in minutes. These are Apache Sedona with K-D-B-Tree partitioning and R-Tree local indexing (AS KDB-RT), Spatial Spark with 512x512 Sort Tile Partitioning (SP STP), Location Spark with Quad-Tree partitioning and R-Tree local indexing (LS QT-RT) and Magellan with Z-Order Curves of 20 precision. For D_6, Magellan's execution was terminated after 24 hours.

Table 11. Total wall-clock execution time over all datasets in Table 6 in minutes.

	D_1	D_2	D_3	D_4	D_5	D_6
AS KDB-RT	14.06	**15.05**	41.95	185.79	**73.20**	**28.12**
SP STP	18.60	29.34	62.14	163.96	134.01	424.70
LS QT-RT	**11.07**	16.19	**36.80**	**122.49**	79.36	39.68
Magellan 20	17.29	33.73	68.38	409.94	1064.60	>1440.00

The best performing frameworks are Location Spark and Apache Sedona. Location Spark is performing better for datasets D_1, D_3 and D_4. We expected Location Spark to be superior among all the datasets since the volume of data is enormous and it employs a skew analysis. However, K-D-B-Tree partitioning distributes the input data in a more effective way than Quad Tree partitioning for some datasets, hence Apache Sedona's better performance in D_2, D_5 and D_6. It must also be noted that Magellan's execution time skyrockets from D_4 on, due to Magellan's preprocessing. We observed that during that phase, Magellan assigned geometries in significantly more partitions than the rest of the frameworks, resulting in high overhead in the filtering phase. Spatial Spark's Sort Tile partitioning is on par with Apache Sedona and Location Spark, excluding dataset D_6. However, as explained above, a new sampling technique must be implemented, because it produced wrong verification results.

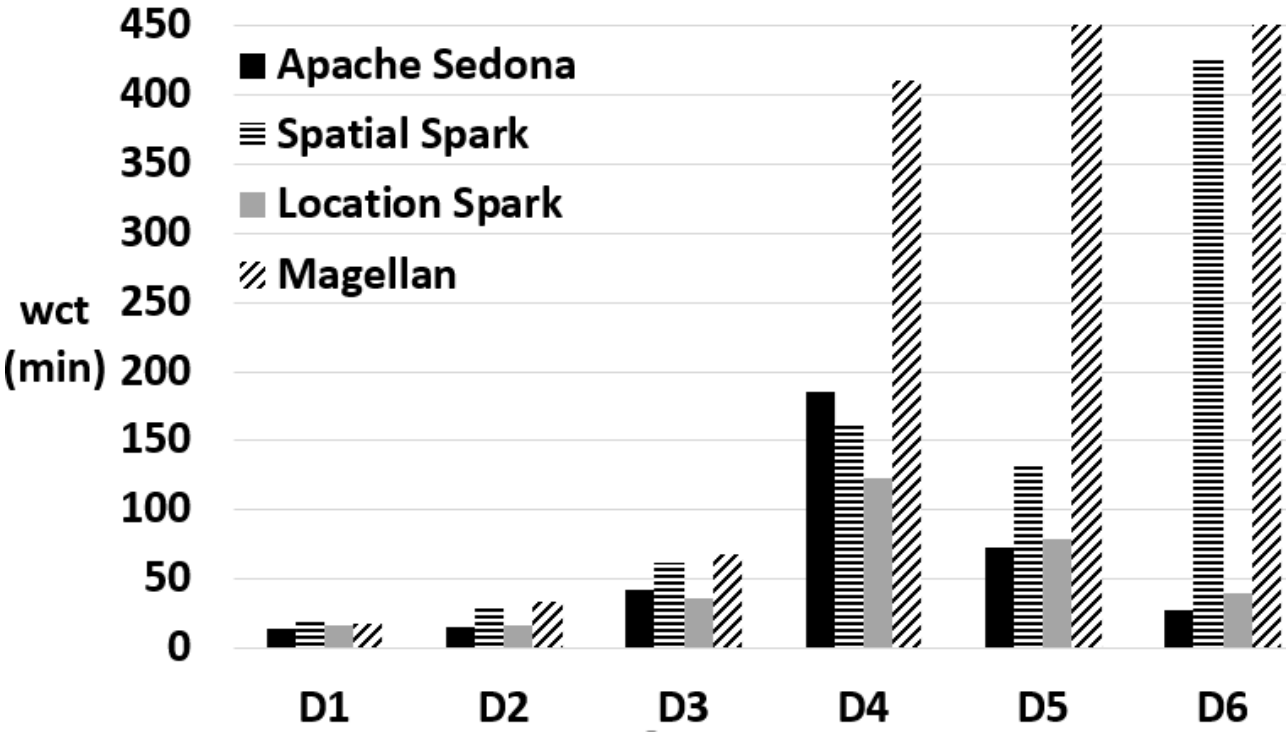

Figure 45. Line chart for total execution timTe of D1-D6.

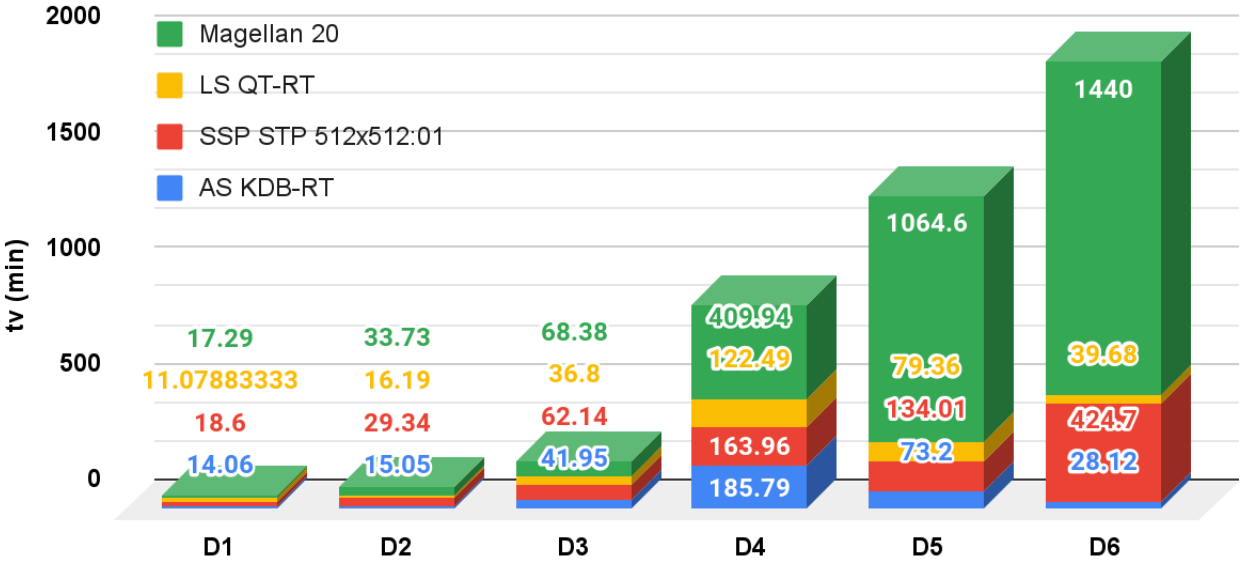

Figure 46. Stacked column chart for total execution time of D1-D6.

6. CONCLUSIONS AND FUTURE WORK

To conclude, the purpose of this thesis is to gather famous serial algorithms and parallel frameworks into a single, novel tool for geospatial interlinking: JedAI-spatial. We have categorized the spatial join algorithms in three dimensions; space tiling, budget-awareness and execution mode. For the serial methods, we introduced new approaches (Strip Sweep) and optimized the implementation of the existing ones (e.g., see the implementation improvements for RADON) For the parallel frameworks, which run on top of Apache Spark, we enhanced their filtering process with the Reference Point Method for duplicate elimination. We performed a thorough evaluation that involves six real, voluminous datasets as well as all serial and parallel budget-agnostic (i.e., batch) algorithms with the results providing interesting insights into their relative performance.

JedAI-spatial is a robust system but some of its methods need tweaking. Algorithms such as CR-Tree and the partition-based ones that use list sweep structures need further optimization. We believe CR-Tree performs below expectations due to the fact that it is implemented in Java, which does not allow proper memory manipulation compared to programming languages like C or C++. Spatial Spark's Binary Split partitioning needs optimization, as well as a superior policy for declaring the sampling ratio for Sort Tile partitioning. R-Tree partitioning for the preprocessing phase is simply not compatible with the Reference Point Method due to its overlapping partitions. For that reason, we plan to integrate R-Grove [30], a method that addresses all our concerns analyzed in subsection 4.3.1.

Apart from tweaking the already improved methods, we aim to integrate other famous algorithms, spatial structures and bulk load techniques. Considering the remarkable execution time for Quad Tree in section 5.1., we believe that similar tree-based algorithms, like k-d Tree [34], should be included in our system. Another useful addition would be STR packing, which as afore-mentioned we believe can solve a lot of problems for some datasets with overlapping geometries.

The main advantage of JedAI-spatial, besides the benchmarking and the categorization of the established algorithms for Geospatial Interlinking, is that it offers a common platform for introducing improvements to all supported techniques.

For example, we plan to develop a novel load-balancing method that enhances all parallel algorithms, since their bottleneck typically lies in the partitioning phase. Recall subsection 5.2.3, where Magellan's execution skyrockets, due to inferior partitioning which extended its filtering phase. Using the skew analysis strategies of Location Spark as a stepping stone, we could develop a powerful, yet generic load-balancer that distributes the workload evenly, among all available workers.

Similarly, we are currently working on a novel algorithm that replaces the typical, coarse-grained MBR of each geometry with another one of finer granularity. This approach will allow for reducing the false positives of FIltering to a significant extent. These are candidate pairs that have intersecting MBRs but satisfy no topological relation apart from `disjoint`. We also intend to extend JedAI-spatial in various ways: by adding support for point geometries and proximity relations, by combining it with JedAI [31], [35], [36], [37], [38] to explore the interplay between textual and spatial information in data integration and by publishing as a Docker-based web application that can be easily used for free. This practice was followed in JedAI [31], [35], [36], [37], [38], which provided an intuitive Graphical User Interface (GUI) that allowed users to manually configure their desired entity resolution methods. JedAI-spatial's web application would allow users to provide input datasets, configure their spatial join algorithms and save their results in RDF format. Moreover, as we already mentioned we hope to release two versions, one for parallel and

one for serial implementations, as Maven, Gradle or sbt[25] dependencies. Since JedAI-spatial is open-source under Apache License V2.0, we encourage other engineers to submit their own spatial join algorithms and frameworks so as to build a toolkit with various techniques.

Finally, we plan to apply JedAI-spatial to a wealth of real-world applications, such as the interlinking of satellite images with in-situ observations [32] or news articles extracted from the Web [33].

[25] https://www.scala-sbt.org/

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Semantic Web	Σημασιολογικός Ιστός
Geospatial Interlinking	Διασύνδεση Γεωχωρικών Δεδομένων
Spatial Join	Χωρική Διασύνδεση

ABBREVIATIONS - ACRONYMS

IoT	Internet of Things
DE-9IM	Dimensionally Extended 9-Intersection Model
RAM	Random-Access Memory
CPU	Central Processing Unit
AWS	Amazon Web Services
RDD	Resilient Distributed Dataset
OGC	Open Geospatial Consortium
JTS	Java Topology Suite
WKT	Well-Known Text
CSV	Comma-Separated Values
TSV	Tab-Separated Values
RDF	Resource Description Framework
MBR	Minimum Bounding Rectangle
RPM	Reference Point Method
PBSM	Partition Based Spatial-Merge Join
SSSJ	Scalable Sweeping-Based Spatial Join
CR-Tree	Cache-Consciours R-Tree
STR	Sort Tile Recursive
RMBR	Relative Minimum Bounding Rectangle
QRMBR	Quantized Relative Minimum Bounding Rectangle
NE	NorthEast
NW	NorthWest
SE	SouthEast
SW	SouthWest
CF	Co-occurrence Frequency
JS	Jaccard Similarity
MBRO	Minimum Bounding Rectangle Overlap
ISP	Inverse Sum of Points
HDFS	Hadoop Distributed FileSystem
SRDD	Spatial Resilient Distributed Dataset
SQL	Structured Query Language
API	Application Programming Interface

YAML	Yet Another Markup Language
GUI	Graphical User Interface

ANNEX I

The code for the Apache Spark-based implementations is open-source and available at https://github.com/GeoLinker/GeoLinker.

In order to run an experiment on the cluster, one has to first build a *fat jar (Java Archive)* file using the command `sbt assembly` and provide the configuration for the execution according to the configuration file at `config/configurationTemplate.yaml`.

A sample execution for a Magellan experiment would be the following:

```
$ sbt assembly
$ spark-submit --master <master> --class experiments.MagellanExp
target/scala-2.11/DS-JedAI-assembly-0.1.jar <options> -conf
</path/to/configuration.yaml>
```

Other experiments that are available:

```
experiments.GeoSparkExp
experiments.SedonaExp
experiments.SpatialSparkPartitionedExp
experiments.SpatialSparkExp
experiments.LocationSparkExp
```

Hereby, we present sample configuration files for each method, which we used in our experiments for the D_1 dataset:

GeoSpark:

```
source:
  path: "hdfs:///user/gman/ds-jedai/AREAWATER.tsv"
  realIdField: "2"
  geometryField: "0"

target:
  path: "hdfs:///user/gman/ds-jedai/LINEARWATER.tsv"
  realIdField: "2"
  geometryField: "0"

relation: "DE9IM"

configurations:
  partitions: "400"
  magellanZOrderCurvePrecision: "20"
```

Apache Sedona:

```
source:
  path: "hdfs:///user/gman/ds-jedai/AREAWATER.tsv"
  realIdField: "2"
  geometryField: "0"

target:
  path: "hdfs:///user/gman/ds-jedai/LINEARWATER.tsv"
  realIdField: "2"
  geometryField: "0"

relation: "DE9IM"

configurations:
  sedonaGridType: "KDBTREE"
  sedonaLocalIndexType: "RTREE"
  partitions: "400"
```

Location Spark:

```
source:
  path: "hdfs:///user/gman/ds-jedai/AREAWATER.tsv"
  realIdField: "2"
  geometryField: "0"

target:
  path: "hdfs:///user/gman/ds-jedai/LINEARWATER.tsv"
  realIdField: "2"
  geometryField: "0"

relation: "DE9IM"

configurations:
  locationSparkLocalIndexType: "QUADTREE"
  partitions: "400"
```

Magellan:

```
source:
  path: "hdfs:///user/gman/ds-jedai/AREAWATER.tsv"
  realIdField: "2"
  geometryField: "0"
```

```
target:
  path: "hdfs:///user/gman/ds-jedai/LINEARWATER.tsv"
  realIdField: "2"
  geometryField: "0"

relation: "DE9IM"

configurations:
  partitions: "400"
  magellanZOrderCurvePrecision: "20"
```

Spatial Spark Partitioned:

```
source:
  path: "hdfs:///user/gman/ds-jedai/AREAWATER.tsv"
  realIdField: "2"
  geometryField: "0"

target:
  path: "hdfs:///user/gman/ds-jedai/LINEARWATER.tsv"
  realIdField: "2"
  geometryField: "0"

relation: "DE9IM"

configurations:
  partitions: "400"
  spatialSparkMethod: "FGP"
  spatialSparkMethodConf: "512:512"
```

REFERENCES

[1] "What is geospatial data?," *Ibm.com*. [Online].
Available: https://www.ibm.com/topics/geospatial-data. [Accessed: 16-Oct-2021].

[2] "Uber Hits 5 Billion Rides Milestone," *Uber.com*, 29-Jun-2017. [Online].
Available: https://www.uber.com/en-SG/blog/uber-hits-5-billion-rides-milestone/.
[Accessed: 16-Oct-2021].

[3] A.-C. Ngonga Ngomo, "ORCHID – reduction-ratio-optimal computation of Geo-spatial distances for link discovery," in *Advanced Information Systems Engineering*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 395–410.

[4] C. Strobl, "Dimensionally extended nine-intersection model (DE-9IM)," in *Encyclopedia of GIS*, Boston, MA: Springer US, 2008, pp. 240–245.

[5] E. P. F. Chan and J. N. H. Ng, "A general and efficient implementation of geometric operators and predicates," in *Advances in Spatial Databases*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 67–93.

[6] B. Sowell, M. V. Salles, T. Cao, A. Demers, and J. Gehrke, "An experimental analysis of iterated spatial joins in main memory," *Proceedings VLDB Endowment*, vol. 6, no. 14, pp. 1882–1893, 2013.

[7] V. Pandey, A. Kipf, T. Neumann, and A. Kemper, "How good are modern spatial analytics systems?," *Proceedings VLDB Endowment*, vol. 11, no. 11, pp. 1661–1673, 2018.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, p. 10.

[9] J.-P. Dittrich and B. Seeger, "Data redundancy and duplicate detection in spatial join processing," in *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, 2002.

[10] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Scalable sweeping-based spatial join," in *Proceedings of the 24rd International Conference on Very Large Data Bases*, 1998, pp. 570–581.

[11] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data - SIGMOD '96*, 1996.

[12] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec*, vol. 14, no. 2, pp. 47–57, 1984.

[13] K. Kim, S. K. Cha, and K. Kwon, "Optimizing multidimensional index trees for main memory access," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data - SIGMOD '01*, 2001.

[14] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta Inform.*, vol. 4, no. 1, pp. 1–9, 1974.

[15] R. H. Güting and W. Schilling, "A practical divide-and-conquer algorithm for the rectangle intersection problem," *Inf. Sci. (Ny)*, vol. 42, no. 2, pp. 95–112, 1987.

[16] S. T. Leutenegger, M. A. Lopez, and J. Edgington, "STR: a simple and efficient algorithm for R-tree packing," in *Proceedings 13th International Conference on Data Engineering*, 2002.

[17] M. A. Sherif, K. Dreßler, P. Smeros, and A.-C. N. Ngomo, “RADON — rapid discovery of topological relations,” in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017, pp. 175–181.

[18] G. Papadakis, G. Mandilaras, N. Mamoulis, and M. Koubarakis, “Progressive, Holistic Geospatial Interlinking,” in *Proceedings of the Web Conference 2021*, 2021.

[19] P. Smeros and M. Koubarakis, “Discovering Spatial and Temporal Links among RDF Data,” in *Proceedings of the Workshop on Linked Data on the Web, LDOW 2016, co-located with 25th International World Wide Web Conference (WWW 2016)*.

[20] S. You, J. Zhang, and L. Gruenwald, “Large-scale spatial join query processing in Cloud,” in *2015 31st IEEE International Conference on Data Engineering Workshops*, 2015.

[21] J. Yu, Z. Zhang, and M. Sarwat, “Spatial data management in apache spark: the GeoSpark perspective and beyond,” *Geoinformatica*, vol. 23, no. 1, pp. 37–78, 2019.

[22] J. Yu, J. Wu, and M. Sarwat, “GeoSpark: A cluster computing framework for processing large-scale spatial data,” in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS ’15*, 2015.

[23] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, “LocationSpark: A distributed in-memory data management system for big spatial data,” *Proceedings VLDB Endowment*, vol. 9, no. 13, pp. 1565–1568, 2016.

[24] M. Tang, Y. Yu, A. R. Mahmood, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, “LocationSpark: In-memory distributed spatial query processing and optimization,” *Front. Big Data*, vol. 3, p. 30, 2020.

[25] G. M. Morton, *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. 1996.

[26] A. Eldawy and M. F. Mokbel, “SpatialHadoop: A MapReduce framework for spatial data,” in *2015 IEEE 31st International Conference on Data Engineering*, 2015.

[27] D. Tsitsigkos, P. Bouros, N. Mamoulis, and M. Terrovitis, “Parallel in-memory evaluation of spatial joins,” in *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL ’19*, 2019.

[28] P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind, *Geographic Information Systems and Science*, 2nd ed. John Wiley & Sons, 2007.

[29] A. F. Ahmed, M. A. Sherif, and A.-C. N. Ngomo, “On the effect of geometries simplification on Geo-spatial link discovery,” *Procedia Comput. Sci.*, vol. 137, pp. 139–150, 2018.

[30] T. Vu and A. Eldawy, “R-Grove: Growing a family of R-trees in the big-data forest,” in *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL ’18*, 2018.

[31] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, and M. Koubarakis, “The return of jedAI: End-to-end entity resolution for structured and semi-structured data,” *Proceedings VLDB Endowment*, vol. 11, no. 12, pp. 1950–1953, 2018.

[32] Desta Haileselassie Hagos, Theofilos Kakantousis, Vladimir Vlassov, Sina Sheikholeslami, Tianze Wang, Jim Dowling, Claudia Paris, Daniele Marinelli, Giulio Weikmann, Lorenzo Bruzzone, Salman Khaleghian, Thomas Krämer, Torbjørn Eltoft, Andrea Marinoni, Despina-Athanasia Pantazi, George Stamoulis, Dimitris Bilidas, George Papadakis, Georgios

M. Mandilaras, Manolis Koubarakis, Antonis Troumpoukis, Stasinos Konstantopoulos, Markus Muerth, Florian Appel, Andrew Fleming, Andreas Cziferszky, “ExtremeEarth meets satellite data from space,” IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens., vol. 14, pp. 9038–9063, 2021.

[33] M. Koubarakis, K. Bereta, G. Papadakis, D. Savva, and G. Stamoulis, “Big, linked geospatial data and its applications in earth observation,” IEEE Internet Comput., vol. 21, no. 4, pp. 87–91, 2017.

[34] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” Commun. ACM, vol. 18, no. 9, pp. 509–517, 1975.

[35] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, M. Koubarakis, “JedAI: The Force Behind Entity Resolution.”, ESWC (Satellite Events): 161-166, 2017.

[36] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, M. Koubarakis, “Domain- and Structure-Agnostic End-to-End Entity Resolution with JedAI.”, SIGMOD Rec. 48(4): 30-36, 2019.

[37] G. Papadakis, G. M. Mandilaras, L. Gagliardelli, G. Simonini, E. Thanos, G. Giannakopoulos, S. Bergamaschi, T. Palpanas, M. Koubarakis, “Three-dimensional Entity Resolution with JedAI.”, Inf. Syst. 93: 101565, 2020.

[38] G. Papadakis, L. Tsekouras, E. Thanos, N. Pittaras, G. Simonini, D. Skoutas, P. Isaris, G. Giannakopoulos, T. Palpanas, M. Koubarakis, “JedAI3 : beyond batch, blocking-based Entity Resolution.”, EDBT: 603-606, 2020.