# Parallel Spatial Join Processing with Adaptive Replication

Nikolaos Koutroumanis
Dept. of Digital Systems
University of Piraeus
Piraeus, Greece
koutroumanis@unipi.gr

Christos Doulkeridis
Dept. of Digital Systems
University of Piraeus
Piraeus, Greece
cdoulk@unipi.gr

Akrivi Vlachou
Dept. of Inf. & Com. Syst. Engineering
University of Aegean
Karlovasi, Greece
avlachou@aegean.gr

## ABSTRACT

Parallel spatial join algorithms are essential for scalable processing and analysis of big spatial data. The state-of-the-art algorithms rely on splitting the data into partitions and replicating objects from one data set in neighboring partitions, so that partitions can be processed in parallel independently without producing duplicate results. However, this universal replication of one data set leads to suboptimal performance in the case of skewed data sets with varying density. Instead, we advocate an approach that adaptively selects which data set to replicate in different local areas of the space, thus minimizing replication and boosting the performance of query processing. To this end, we contrive a graph-based framework for modeling replication between neighboring partitions. We study the theoretical properties that lead to adaptive replication with correct and duplicate-free results. Then, we design a data-parallel algorithm in Apache Spark which is based on adaptive replication, and we demonstrate its performance gain over the state-of-the-art for large-sized data sets, real and synthetic, under various settings.

## 1 INTRODUCTION

Parallel spatial joins [2, 3, 10, 13, 14, 17, 21, 27, 28] has been a field of great interest in the era of big spatial data [7], as they find application in diverse scientific domains, including urban planning, cartography, neuroscience and astrophysics. In this paper, we revisit a fundamental problem in spatial joins, namely the $\epsilon$-distance spatial join $R \bowtie_\epsilon S$ between point data: given two collections of spatial points $R$ and $S$, find the pairs of points $(r, s)$ with $r \in R$, $s \in S$, such that their distance $d()$ is smaller than $\epsilon$, i.e., $d(r, s) \leq \epsilon$.

Existing algorithms for parallel spatial joins are based on splitting the data points into partitions and replicating points from one data set locally to neighboring partitions [13, 20, 22, 24, 25, 27]. In this way, partitions can be processed independently in parallel, and the algorithm guarantees correctness without duplicate results. As indicated in a recent comparative study of spatial join algorithms [12], one of the best performing and widely-used algorithms for in-memory spatial joins is PBSM [13], which forms the basis for many other variations of parallel spatial join algorithms.

In PBSM, a grid is used for splitting the space into cells (partitions) and each point is assigned to the cell that encloses it. Also, each point of *one of the two data sets*, e.g., $r \in R$, is additionally replicated to cells that intersect with the circle centered at $r$ with radius equal to the distance join threshold $\epsilon$. The concept is illustrated in Figure 1a, where $r_3$ must also be replicated to the upper neighboring cell, so that the join result $(r_3, s_3)$ can be identified by examining the points of the upper partition only. It is trivial to show that after replication, PBSM guarantees that the correct result without duplicates, can be computed by examining individual cells.

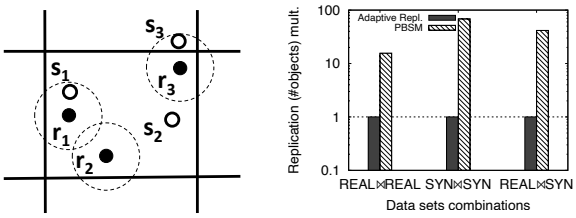



**Figure 1: Example of point replication (left) and relative overhead in replication of PBSM over our approach (right).**

Nevertheless, the global selection of a single data set for replication leads to suboptimal performance in practice, especially in the case of skewed data sets with varying density. Figure 1b shows the relative overhead (in log scale) in terms of number of replicated objects of PBSM compared to the *adaptive replication* approach proposed in this paper. As shown in the chart, for different combinations of data sets, the adaptive replication that makes local decisions about which data set to replicate results can reduce replication by a factor of 10x–75x, while guaranteeing result correctness. In turn, this gain by reduced replication greatly improves the performance of query processing.

Motivated by this observation, in this paper, we introduce a novel approach for parallel spatial joins, which exploits *adaptive replication* of points to neighboring partitions. Essentially, our approach allows neighboring partitions to make local decisions about the data set that will be replicated, so as to minimize replication and (consequently) processing costs, without compromising correctness nor causing duplicate results. This is particularly beneficial in the case of non-uniform or skewed data sets of varying density, which are commonly encountered in real applications. As a result, our approach manifests as a data-parallel algorithm that is more efficient than the state-of-the-art for parallel spatial joins.

Concretely, in this paper, we make the following contributions:

- We identify a limitation of state-of-the-art algorithms for parallel spatial joins, namely excessive data replication, which has negative effect on performance.
- We contrive a graph-based framework for modeling neighboring data partitions, leading to theoretical properties for the design of adaptive object replication such that the corresponding join algorithm is correct and duplicate-free.
- We design and implement an efficient algorithm for parallel spatial joins in Apache Spark that outperforms the state-of-the-art.
- By means of experiments on real and synthetic data sets, we demonstrate the performance gain of our approach,

in comparison with PBSM and Apache Sedona, a state-of-the-art framework for big spatial data processing.

The rest of the paper is structured as follows: Section 2 reviews related work. In Section 3 we provide the problem formulation along with useful properties, and we motivate our approach. Then, in Section 4, we present a graph-based abstraction, called the graph of agreements, and its background. Section 5 provides the details on adaptive replication of objects. Section 6 describes the proposed data-parallel algorithm for spatial joins in Apache Spark. In Section 7, we present the results of the experiments and we conclude in Section 8.

## 2 RELATED WORK

**Distributed/Parallel Algorithms.** Parallel spatial join algorithms are classified as *single-assigned multi-join* (SAMJ) or *multi-assigned single-join* (MASJ), based on object assignment and how partitions are joined. In SAMJ, an object is assigned to a single partition, and a partition from the one set may be joined with multiple partitions from the other set. Instead, in MASJ, an object may be assigned to more than one partitions, but then each partition is joined with a single other partition. MASJ methods have the property of producing duplicate results, which has a negative impact on performance due to communication cost and the extra workload. As such, a deduplication process is required after the join computation or the use of the reference-point duplicate avoidance technique [5].

In the SAMJ category, one of the first algorithms utilize the R-Tree structure [3]. The join is performed on two R-Trees in which the overlapping root entries that constitute a sub-tree, are joined. In the MASJ category, the *Partition-based Spatial Merge Join* (PBSM) algorithm [13] partitions the space uniformly into disjoint cells of equal size (grid partitioning), and the objects of the two data sets are assigned to one or more cells (so as to ensure correctness). Then, cells are assigned to processors based on a strategy (e.g., round-robin). Interestingly, as indicated in a recent comparative study of spatial join algorithms [12], PBSM [13] is one of the best performing algorithms for in-memory spatial joins. A similar approach to PBSM, but for parallel processing, is proposed in [28]. Later, Patel and DeWitt [14] highlight the adverse effect of data skewness. Partitions are distributed to workers by using a hash function or in a round-robin scheme, and two parallel join algorithms are introduced, the *Clone* and the *Shadow* join.

SPINOJA [17] introduces the MOD-Quadtree (metric-based object decomposition Quadtree), where the criterion for subdividing a cell is a work metric (not the number of objects). Later, an adaptive partitioning algorithm (ADP) [23] is proposed to form load-balanced workload partitions of spatial objects with extent, such as polygon and polyline. Both studies employ duplicate avoidance techniques. Lately, Tsitsigkos et. al. [21] revamp the popular PBSM algorithm, targeting to optimize its execution through the proper tuning of the number of grid partitions and the axis on which plane sweep will be performed per partition. Parameter tuning has been also studied in [19] with implementation aspects, where substantial performance gain is achieved with a uniform grid against other structures such as R-Tree and CR-Tree. In TRANSFORMERS [15], disk-based spatial joins are executed in a data-driven manner. The algorithm detects divergence in densities, and adjusts its execution accordingly.

**MapReduce Algorithms.** In recent years, many studies leverage the MapReduce paradigm [4] for processing big spatial data. In SJMR (*Spatial Join With MapReduce*) [27], grid partitioning is applied to form $N$ tiles that are assigned to $P$ partitions in a round-robin manner. In the map phase, every object is assigned to one or more partitions, while in the reduce phase, the space of each partition is split to stripes according to tiles and plane sweep is performed for each tile. García-García et al. [10] present and evaluate algorithms for distance-based join queries and closest pairs queries in SpatialHadoop [6]. A later version of the work improves the efficient of the distance-join query using Voronoi-based partitioning [8]. MR-DSJ [18] adjusts the size of the grid cells and adopts a rule-based replication approach among cells that guarantees correct and duplicate-free results, without considering the data distribution. Pechlivanoglou et al. [16] propose a distributed sweep-line algorithm for spatial intersection joins, implemented in Apache Spark [26]. Apache Sedona (formerly GeoSpark) [25], LocationSpark [20], SpatialSpark [24] and Simba [22] are engines that extend Spark, supporting distributed spatial query operations and indexes. In [9], $k$ nearest neighbor join query and $k$ closest pairs query are implemented on top of Apache Sedona, LocationSpark and Simba, and a comparative study on different parameters is conducted regarding their performance.

## 3 PRELIMINARIES

### 3.1 Problem Statement

Consider two sets of spatial points $R$ and $S$, where each point $r \in R$ and $s \in S$ is described by its coordinates $(r.x, r.y)$ and $(s.x, s.y)$ respectively. The problem of the $\epsilon$-distance spatial join is to retrieve pairs $(r, s)$ that are within a user-specified distance threshold $\epsilon$.

*Definition 3.1.* ($\epsilon$-distance spatial join) Given two collections of spatial points $R$ and $S$ the $\epsilon$-distance join ($R \bowtie_\epsilon S$) identifies the pairs $(r, s)$ with $r \in R$, $s \in S$, such that $d(r, s) \leq \epsilon$, where $d(r, s)$ is a distance function and $\epsilon$ is the distance threshold of the join.

In its simplest version, the computation of spatial distance join can be performed by comparing all points of $R$ with all points of $S$. The cost is $O(|R| \cdot |S|)$ where $|R|$ and $|S|$ denote the cardinalities of the two data sets respectively. Even though this cost may be tolerable for small data sets, it becomes prohibitively expensive for large data sets. Therefore, spatial joins are addressed by a *divide-and-conquer* strategy, where the computation is broken in smaller parts [13]. To achieve that, space partitioning methods are utilized that group the data into a set of partitions $\{P_1, P_2, ..., P_n\} \in P$ that contain spatial points. In the case of grid partitioning, which is used in this paper, the partitions correspond to grid cells. Then, we are able to compute the result set of the spatial join operation, by processing the join in each individual partition.

The remaining question is how to assign points to partitions. We identify two important properties for the assignment of points to partitions that are relevant for any $\epsilon$-distance spatial join algorithm.

*Definition 3.2.* (correctness) In the $\epsilon$-distance spatial join, an assignment of points to $n$ partitions is called correct, if the result of the join can be derived from the union of the results of each partition, i.e., $R \bowtie_\epsilon S = \bigcup R_i \bowtie_\epsilon S_i$, where $i = 1 \ldots n$.

*Definition 3.3.* (duplicate-free assignment) In the $\epsilon$-distance spatial join, an assignment of points to $n$ partitions is called duplicate-free, if the intersection of the results of all partitions is empty, i.e., $\bigcap R_i \bowtie_\epsilon S_i = \emptyset$, where $i = 1 \ldots n$.

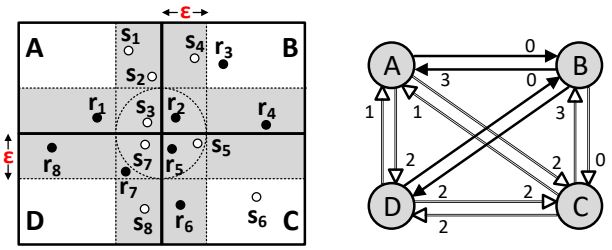



**Figure 2: Running example with four cells.**

## 3.2 Motivation and Objective

Consider Figure 2 showing a data space partitioned in four cells $\{A, B, C, D\}$ in order to create the workload partitions for computing the $\epsilon$-distance spatial join. Assume that the sides of each cell are greater than $2 \cdot \epsilon$. Points in the grey region are candidates for replication. Applying PBSM would require choosing one of the two data sets (say $R$) and then replicating its points $r \in R$ to any cell $c_i$ that is within distance $\epsilon$. We use $MINDIST(r, c_i)$ to denote the minimum distance of a point $r$ to any point of cell $c_i$, thus the condition for replication of $r$ to $c_i$ is $d(r, c_i) \leq \epsilon$.

Table 1 shows for the example of Figure 2 the replicated points for each cell and the cell to which they are replicated. In the table, the worst-case cost per cell is shown, as the product of the number of the two sets of points in a cell. The cost is equivalent to the number of comparisons $(r \cdot s)$ for performing the spatial join operation $R_i \bowtie_\epsilon S_i$ in the $i$-th cell. In this example, by replicating the points from the R set universally, we can achieve a lower overall cost of spatial join compared to the universal replication of S set (41 < 42) with a lower number of replicated objects (12 < 13) and with a lower maximum cost per partition (15 < 18).

Universal replication of $R$ set

| Replicated to | Cells | Replicated from A | Replicated from B | Replicated from C | Replicated from D | Cost per cell ($r \cdot s$) |
|---|---|---|---|---|---|---|
| | $A$ | - | $r_2$ | $r_5$ | $r_7, r_8$ | (1+4)·3 = 15 |
| | $B$ | $\emptyset$ | - | $r_5$ | $\emptyset$ | (3+1)·1 = 4 |
| | $C$ | $\emptyset$ | $r_2, r_4$ | - | $r_7$ | (2+3)·2 = 10 |
| | $D$ | $r_1$ | $r_2$ | $r_5, r_6$ | - | (2+4)·2 = 12 |

Universal replication of $S$ set

| Replicated to | Cells | Replicated from A | Replicated from B | Replicated from C | Replicated from D | Cost per cell ($r \cdot s$) |
|---|---|---|---|---|---|---|
| | $A$ | - | $s_4$ | $s_5$ | $s_7$ | 1·(3+3) = 6 |
| | $B$ | $s_1, s_2, s_3$ | - | $s_5$ | $s_7$ | 3·(1+5) = 18 |
| | $C$ | $s_3$ | $\emptyset$ | - | $s_7, s_8$ | 2·(2+3) = 10 |
| | $D$ | $s_3$ | $\emptyset$ | $s_5$ | - | 2·(2+2) = 8 |

**Table 1: The replicated objects and the respective cost per cell when replicating universally the $R$ and the $S$ set.**

The objective in this paper is to design an algorithm that avoids the need for universal replication of one data set, while maintaining the desired properties of correctness and duplicate-free assignment. The algorithm will choose a different set for replication in different areas of the space in an adaptive way based on the data distribution. Thus, our objective is to minimize the replicated points, and consequently improve the performance by minimizing execution time. However, the main challenge when alternating between data sets for replication is to maintain the duplicate-free property. For example, assume that $r_2$ joins with $s_5$ in Figure 2, as they are within distance $\epsilon$. Since cells $A$ and $B$ exchange points from $R$ set, $r_2$ will be replicated from cell $B$ to $A$. Also, since other cells exchange objects from $S$ set, $s_5$ will be replicated from cell $C$ to $A$ and $B$. Thus, the pair $\{r_2, s_5\}$ will be reported by both $A$ and $B$ cells, leading to duplicate results. Existing algorithms for parallel spatial joins do not address this challenge and this is the novelty of our approach.

# 4 AGREEMENT-BASED REPLICATION

In this section, we present a graph-based abstraction for modeling the replication between neighboring grid cells. In contrast to PBSM, where one of the data sets is chosen universally for replication between any pair of adjacent cells, our method follows an adaptive approach. Essentially, a pair of adjacent cells may take different decision (called *agreement*) than another pair of cells, about which data set to replicate. In this way, we achieve *adaptive replication* of points to partitions (Sect. 5) and we set the ground for an efficient parallel spatial join algorithm (Sect. 6).

## 4.1 Grid Resolution

Consider a regular grid in the 2-dimensional data space that consists of equi-sized cells. In the case of spatial joins, fine-grained cells can lead to excessive data replication, while coarse-grained cells can lead to partitions that cover large spaces, thus producing a large number of candidate pairs of points.

To address this issue, we construct cells whose side length $l$ is at least twice larger than the distance threshold $\epsilon$, i.e., $l > 2 \cdot \epsilon$. In this way, we ensure that a point will be replicated to at most 3 other cells, thus imposing a limit to amount of replicated data, and restricting the number of candidate objects that are produced by each cell. The number $m_x$ (resp. $m_y$) of cells for dimension $x$ (resp. $y$) is computed as: $m_x = \lceil (x_{max} - x_{min})/2 \cdot \epsilon) \rceil - 1$, where $x_{min}, x_{max}$ correspond to the minimum and maximum values in the data set for $x$.

## 4.2 Graph of Agreements

We apply an approach that is based on selective replication of $R$ and $S$ data sets, for different cells, by introducing the notion of *agreements* between cells. Agreements can be utilized in grid structures that fulfill the condition $l > 2 \cdot \epsilon$. An agreement designates the type of the data set (i.e., $R$ or $S$) that will be replicated between two adjacent cells (i.e., those having at least one common point), and it is bilaterally adhered.

We introduce an abstraction, named *graph of agreements*, to model agreements between cells. Each vertex of the graph represents a cell. Two vertices $v_i$ and $v_j$ that represent adjacent cells, are connected with either 1 or 2 pairs of $\{e_{ij}, e_{ji}\}$ edges. A directed edge $e_{ij}$ indicates the type of the data set that will be replicated from vertex $v_i$ to vertex $v_j$. This is defined by its type $\tau$ ($\tau \in \{R, S\}$). The edges that connect 2 particular vertices, have *always* the same type $\tau$. This means that points from the same data set $\tau$ are replicated between these cells. We call this an *agreement* between these cells, and use $\alpha_\tau$ to denote the *agreement type*.

The reason of having 2 pairs of $\{e_{ij}, e_{ji}\}$ edges between $v_i$ and $v_j$ vertices is that these vertices belong to 2 different subgraphs. The graph of agreements is viewed as a set of *subgraphs* composed of 4 fully-connected vertices that represent 4 cells that are adjacent. Hereafter, we will refer to such cells as *quartets*. In the scope of the subgraph, 2 vertices are always connected with 2 edges that belong exclusively to the subgraph. In total, 12 edges exist per subgraph. A vertex may be part of 1, 2 or 4 subgraphs.

*Example 4.1.* Figure 3 depicts an instance of a graph of agreements of a 2×3 grid. Cells are depicted as vertices and the edges

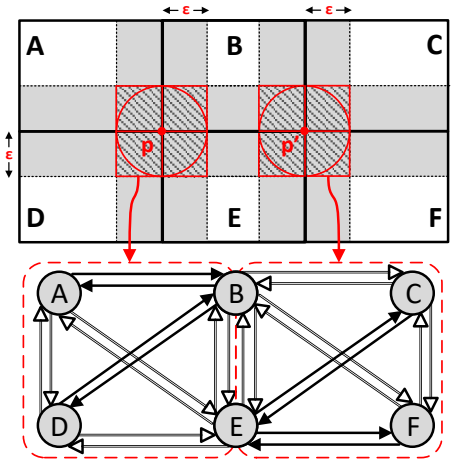



**Figure 3: An instance of grid and the respective graph of agreements that consists of two subgraphs.**

indicate the agreements. The agreement type is illustrated using dark or white coloured edges. Two subgraphs exist in this graph of agreements, namely {A, B, E, D} and {B, C, F, E}, shown with red-coloured dashed borders. Note that vertices B and E are connected with two pairs of edges $\{e_{BE}, e_{EB}\}$, where the two pairs belong to different subgraphs.

*Definition 4.2.* The Graph of Agreements $G(V, E)$ is a directed, weighted multigraph, where:

- each vertex $v \in V$ corresponds to a grid cell.
- two vertices $v_i, v_j$ are linked with one or two pairs of directed edges $\{e_{ij}, e_{ji}\}$.
- a directed edge $e_{ij} \in E$ linking adjacent cells $i$ and $j$ can be of two types.
- the edges that link two vertices are always of the same type.
- $w() : V \times V \rightarrow \mathbb{N}$ is a function that assigns weights to edges.
- the directed edges that have the same tail and head vertices, have the same weight $w$.

## 4.3 Instantiating the Graph of Agreements

Given a grid structure, we can derive many different instances of the graph of agreements, depending on the type of agreement ($\alpha_R$ or $\alpha_S$) between any two adjacent cells. The interest lies on how to select an instance from the pool of potential instances of graph of agreements.

**Determining the agreement type.** Ideally, for a pair of adjacent cells, the agreement type should be of that type $\tau$, which induces the lowest overhead in terms of replication and cost per cell. Based on this, we employ two variants, called LPiB and DIFF, for deciding the agreement type between two adjacent cells.

- *Least points in boundaries (LPiB)*: The agreement type between two adjacent cells is defined by the data set having the minimum number of candidate points for replication between the two cells.
- *Least points in the cell with the greatest difference in number of points (DIFF)*: The cell with the greatest difference in the number of points of the two data sets determines the agreement type. The type of agreement corresponds to the data set that has the fewest points within the cell.

*Example 4.3.* Consider the cells A and D of the grid depicted in Figure 2. When the LPiB approach is applied for choosing the type of agreement between the two cells, we consider the replication area of the two cells, where there exist 2 points of S type ($s_3, s_7$) and 3 points of R type ($r_1, r_7, r_8$). The type with the minimum number of candidate points for replication is the S set (2<3), thus the agreement type is $\alpha_S$. Instead, when the DIFF approach is applied, we consider at first the cell the greatest difference of the number of points from the two sets ($|\#pts_r - \#pts_s|$). Cell A has the greatest difference ($|1-3| = 2$) compared to cell D ($|2-2| = 0$). Thus, the agreement type is $\alpha_R$ as in cell A the fewest points are of R type (not shown in the graph of Figure 2, as it is LPiB-based instantiated).

**Defining edge weights.** After the agreement types have been decided, weights are computed for the edges of the graph. If the agreement between cells $i$ and $j$ is $a_R$, then the edge $e_{ij}$ indicates that R points from cell $i$ will be replicated to cell $j$. The edge weight $w_{ij}$ is the product of the number of points (of type R) that will be replicated from cell $i$ with the number of points (of type S) that exist in cell $j$. Intuitively, the weight $w_{ij}$ shows the processing cost that is caused due to the replication of points from cell $i$ to $j$. This cost corresponds to the number of pairs that need to be examined due to the replication.

*Example 4.4.* In the example of Figure 2, the weight of the edge $e_{BA}$ (of type $a_R$) is the product of the number of R points that will be replicated from cell B to cell A ($r_2$), with the number S points in cell A ($s_1, s_2, s_3$). Therefore $w_{BA} = 1 \cdot 3 = 3$. Instead, edge $e_{CB}$ is of type $a_S$, therefore $w_{CB} = 1 \cdot 3 = 3$ because one S point ($s_5$) will be replicated to cell B, which contains three R points ($r_2, r_3, r_4$).

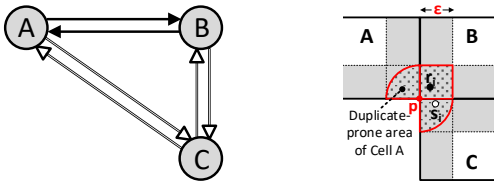
## 4.4 Correctness

Interestingly, the abstraction of the graph of agreements is helpful to provide a deeper understanding of the problem of parallel spatial joins. First, we make the observation that PBSM corresponds to an instance of the graph of agreements, where all agreement types are identical (either $\alpha_R$ or $\alpha_S$). The following lemma justifies the need of having edges of the same type between two nodes, as otherwise the point assignment to cells may jeopardize correctness.

Lemma 4.5. *If the two edges of an agreement are of different type ($\tau$), then the property of correctness cannot be guaranteed.*

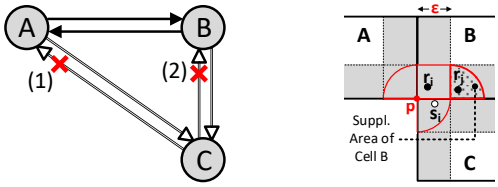
Proof. By contradiction. Let us assume that the property of correctness is guaranteed in the instance of Figure 2, although two edges of an agreement are of different type. Consider two such edges $e_{AB}$ of type $a_R$ and $e_{BA}$ of type $a_S$ between the neighboring cells $A$ and $B$. Further, consider the two points $s_3 \in A$ and $r_2 \in B$ that constitute a join result, i.e., $d(s_3, r_2) \leq \epsilon$. Since $e_{AB}$ is of type $a_R$, only points of data set $R$ will be replicated to cell $B$, thus $s_3$ will not be replicated in cell $B$. Also, since $e_{BA}$ is of type $a_S$, $r_2$ will not be replicated in cell $A$. As a result, the pair $(s_3, r_2)$ will not be assigned to any cell, and thus will not be reported as join result, which contradicts the assumption that correctness is guaranteed. □

Corollary 4.6. *The graph of agreements specifies an assignment that has the property of correctness.*




(a) Graph of agreements (b) Data space

**Figure 4: Graph of agreements producing duplicate results.**




(a) Edge marking: $e_{CA}$ or $e_{CB}$ (b) Supplementary area of cell B

**Figure 5: Triangle ABC with two possible edge markings and the supplementary area of cell B.**

Even though the graph of agreements guarantees correctness, it does not ensure the duplicate-free property. Consider the data space case of Figure 4b. The three depicted cells have a common touching point $p$ and one of them shares borders with the other two. Cell B shares its borders with cells A and C, while A touches with cell C. Let us discern for each space the dark-shaded areas (one per cell) with the red-coloured borders. A point located in these areas is candidate for replication in two cells. Depending on the layout of the three cells, these areas may be quadrant-shaped (as in cell A and C) or squared-shaped (as in cell B). We will refer to these areas as *duplicate-prone areas*, designated in the context of three cells, i.e., a *triad* of cells.

*Definition 4.7.* (duplicate-prone area) A point $o \in c_i$ is located in the duplicate-prone area of $c_i$ in a triad of cells $c_i$, $c_j$, $c_k$, if $MINDIST(o, c_j) \leq \epsilon \wedge MINDIST(o, c_k) \leq \epsilon$.

Points in the duplicate-prone areas of a triad, are responsible for the violation of the duplicate-free property as they may be matched with the same points in two individual cells. This occurs when the cells represented by the vertices on the graph of agreements contain the two agreement types. We will refer to three fully-connected vertices as *triangle*. The terms triad and triangle will be used interchangeably, depending on the context. Triad is used when it is necessary to focus on the space layout of the three cells, while triangle is used for their representation on the graph of agreements.

Lemma 4.8. *The graph of agreements does not guarantee the duplicate-free property, if there exists a triangle with both agreement types.*

Proof. By contradiction. Assume that there exists a triangle with both agreement types in the graph of agreements, and that the duplicate-free property is guaranteed, i.e., duplicate join results cannot be produced. It suffices to construct an instance of (part of) the graph of agreements that will generate duplicate results. Figure 4a shows such an instance for the ABC triangle for the data space of Figure 4b. The pair of points $(r_i, s_i)$ constitutes a join result, as they are within distance $\epsilon$. Based on the agreements shown in the figure, $s_i$ will be replicated to cells $A$ and $B$, while $r_i$ will be replicated to cell $A$. Thus, the pair $(r_i, s_i)$ will be reported by both cells $A$ and $B$, resulting in a duplicate result. □

## 4.5 Duplicate-free Assignment

To resolve the issue of duplicate results and obtain a duplicate-free graph assignment, we focus on the triangles from which duplicates are produced and prevent their generation. This is achieved by means of excluding points from replication that exist in specific areas (Sect. 4.5.1). At the same time, we consider that the correctness property must be ensured. For this, we replicate additional points that exist in other, supplementary areas (Sect. 4.5.2). Then, we make some adaptations on the replication process in order to be applicable on triangles that are part of a subgraph that models the agreements of a quartet of cells (Sect. 4.5.3).

*4.5.1 Edge Marking in a triangle (plain triad case).* Based on Lemma 4.8, a triangle where both $\alpha_R$ and $\alpha_S$ agreements exist may produce duplicate join results. In the respective triad, the problem stems from the cell that replicates the same type of points to the other two cells, e.g., cell C in Figure 4. The points located in the duplicate-prone area of cell C are replicated to the other two cells, i.e., A and B, and this may cause duplicate results. To prevent this, we need to replicate the points in the duplicate-prone area of C to one cell *only* (A or B). For this purpose, we exclude these points from replication to either A or B cell. Notice that this affects only the duplicate-prone area, i.e., the other points of cell C are replicated.

To handle this, we introduce the concept of *edge marking* on the graph. Given a triangle where both $\alpha_R$ and $\alpha_S$ agreements exist, a marked edge $e_{ij}$ excludes the objects located in the duplicate-prone area of cell $i$ from replication to cell $j$. Notice that there are always two candidate edges for marking in such a triangle.

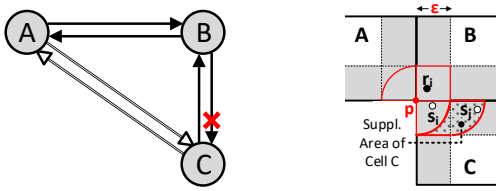
**Edge marking.** The candidate edges for edge marking in a triangle are those having: (i) the same type $\alpha_\tau$, and (ii) the same tail vertex $v_i$ ($e_{ij}$ or $e_{ik}$).

Figure 5a depicts with red cross the two possible edge markings that can occur for the graph of agreements of the example of Figure 4. In the first option, the objects in duplicate-prone area of cell C are excluded from being replicated to cell A, while in the second option, the objects are excluded from being replicated to cell B.

*4.5.2 Ensuring Correction in Edge Marking of a triangle.* In some cases, edge marking has no negative effect on correctness. Specifically, this is the case when the marked edge concerns two cells that have only a single touching point. We capture this in the following corollary.

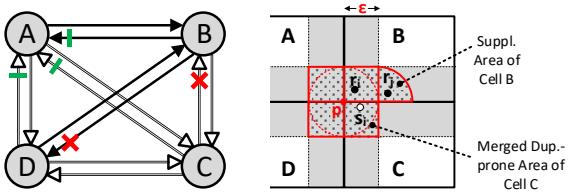
Corollary 4.9. *Edge marking does not violate the correctness property, if the head and tail vertices of the marked edge in a triangle represent cells that have only one common touching point.*

However, in any other case than the one described above, edge marking may affect correctness. Particularly, this is noticed for the marking of the $e_{CB}$ edge in Figure 5a, option (2). For the respective triad (Figure 5b), the points located in the duplicate-prone area of cell C can form pairs with points from an additional subarea of cell B. We will refer to the additional subarea as *supplementary* area. It is depicted as quadrant-shaped and is adjacent to




(a) Graph of agreements (b) Data space

**Figure 6: An instance of a data space with cells A, B, C and its graph of agreements.**




(a) Graph of agreements (b) Grid areas

**Figure 7: Merged duplicate-prone area and edge locking.**

the B's duplicate-prone area. In Figure 5b, $s_i$ will not be replicated to cell B, and thus the pair $(r_j, s_i)$ will not be found in any cell.

The supplementary area of cell $c_j$ in a triad of cells that consist of $c_i$, $c_j$, $c_k$ cells with a common touching point $p$, is defined by the duplicate prone area of $c_i$ whose points will not be replicated to $c_j$. The supplementary area is disjoint from the duplicate-prone area of $c_j$ and a point $o$ located in it, can form pairs with the points in the duplicate-prone area of $c_i$.

*Definition 4.10.* (supplementary area) Given a triad $c_i, c_j, c_k$ with common touching point $p$, a point $o \in c_j$ is located in the supplementary area of cell $c_j$ if: $MINDIST(o, c_i) \leq \epsilon \wedge MINDIST(o, c_k) > \epsilon \wedge d(o, p) \leq 2\epsilon$.

To resolve the matching problem, we consider the points located in the supplementary area as being a supplementary part of its cell's duplicate-prone area. This means that the points in the supplementary area adhere to the replication process of the duplicate-prone area. Specifically, in the example of Figure 5b, the points of R type located in the supplementary area, are replicated to cell A, as the points of the duplicate-prone area of cell B do. In this way, the pair $(r_j, s_i)$ is found in cell A and will be reported as a join result.

As a side-note, the supplementary area may not be quadrant-shaped in all cases. This depends on the shape of the duplicate-prone area of the cell in which the supplementary area exists. Figure 6 shows an instance where the $e_{BC}$ edge in the graph of agreements is marked. For the respective triad in Figure 6b, the supplementary area exists in cell C. The duplicate-prone area of cell C is quadrant-shaped, while the rest illustrated area is the supplementary area.

*4.5.3 Edge Marking in a subgraph's triangles (quartet case).* In the context of a quartet, the area from which duplicates can occur is the *union of all duplicate-prone areas*. This area is depicted as squared-shaped for each cell in Figure 7b. We will refer to it as *merged duplicate-prone area*. Therefore, we need to consider the merged duplicate-prone area of every cell, when we examine the 4 triangles for edge marking.

In a subgraph, correctness can be violated by edge marking in another triangle. For instance, in Figure 7a, $e_{CB}$ is a marked edge based on the ABC triangle. This means that points in the squared-shaped area of cell C (such as $s_i$) are replicated to cell A. Also, points in the duplicate-prone area of cell B (such as $r_i$) and its supplementary area (such as $r_j$) are also replicated to cell A. Therefore, for the ABC triangle, the replication due to the other two edges $e_{CA}$ and $e_{BA}$ is crucial for guaranteeing correctness. However, these edges are eligible for being marked during the examination of another triangle of the subgraph, thus violating correctness. For instance, $e_{BA}$ can be marked during the examination of the ABD triangle. To ensure that such edges *will not be marked* when considering another triangle, we introduce the notion of *edge locking*.

**Edge locking.** In a triangle with vertices $v_i$, $v_j$ and $v_k$, where edge $e_{ij}$ is marked, the edges whose head is the $v_k$ vertex, i.e., $e_{jk}$ and $e_{ik}$, are locked.

In Figure 7a, in the ABC triangle, the marking of edge $e_{CB}$ causes the locking of edges $e_{BA}$ and $e_{CA}$. Edge locking is depicted with a green-colored dash. Hence, in the ABD triangle, only $e_{BD}$ can be marked, as $e_{BA}$ has been locked.

**Duplicate-free property of a grid.** A graph of agreements that represents a grid, does guarantee the duplicate-free property if all of the individual subgraphs that represent the quartets, ensure their duplicate-free property.

## 5 ADAPTIVE REPLICATION

In this section we present the representation of the graph of agreements and how to obtain a duplicate-free assignment, guaranteeing its correctness. Also, we provide the respective algorithms for reaching the duplicate-free assignment.

### 5.1 Representation of Graph of Agreements

We use two dictionaries for the representation of the graph of agreements that allow efficient access to cells and subgraphs. The first dictionary uses the cell identifier as key, and the representation of the cell as value. Thus, we maintain for each cell the number of points from the $R$ and $S$ sets in the areas where points are candidates for replication. This dictionary allows access of the contents of a given cell in constant time. The dictionary is loaded with entries when the sampling procedure of the R and S data sets takes place.

The second dictionary is used to maintain the formed subgraphs. The key in this dictionary is the reference point, i.e., the common touching point of the cells of the quartet. The value is the set of edges associated with the subgraph that consists of these 4 cells. Practically, this set of edges are used to represent the agreements between the respective 4 cells. Thus, the dictionary offers access to the information required about replication in constant time.

### 5.2 Duplicate-free Graph Generation

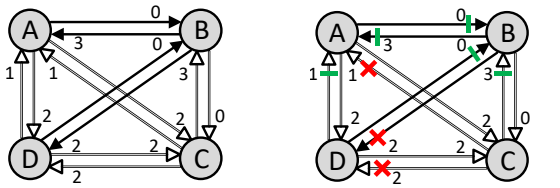
To obtain an assignment that is both correct and duplicate-free, we need to ensure that there exists no subgraph that may produce duplicate results. In turn, this requires that no triangle exists (in any subgraph) that may produce duplicate results.

Thus, we enumerate the subgraphs and for each subgraph we invoke Algorithm 1. Given a subgraph, the algorithm accesses its edges via the dictionary, examines each edge for marking, and if it gets marked then the other edges of the triangle are locked.

**Algorithm 1:** Duplicate-free Graph Generation.

**Input:** Sorted list of edges $E$ of subgraph $s$
**Output:** Processed list of edges $E'$

```
1 E' ← E
2 for each e_ij ∈ E' do
3     if !e_ij.isLocked() then
4         for t_1, t_2 triangles do
5             if e_ik.τ == e_ij.τ and e_jk.τ != e_ij.τ then
6                 if !e_jk.isMarked() and !e_ik.isMarked() then
7                     mark e_ij, lock e_jk, lock e_ik
8                     break
9 return E'
```




**(a) Graph of agreements** **(b) Marked and locked edges**

**Figure 8: Example of applying Algorithm 1.**

As the algorithm needs to access the list of edges, we need to specify an ordering for this list. Intuitively, accessing edges based on descending weight is beneficial, so as to avoid replication of many points. However, this may induce extra replication, due to the supplementary areas. As shown in Section 4.5.2, this occurs only when marking edges whose vertices represent cells with a common side. Consequently, our algorithm prioritizes access to those edges whose two vertices constitute cells with a common touching point, and then proceeds to the rest ones. Therefore, the list of edges contains first the edges whose connected cells have a common touching point and then the edges whose connected cells have a common side, and within each of these groups edges are sorted based on descending weight.

*Example 5.1.* In Figure 8a, the algorithm first traverses every edge that connects two cells with a common touching point ($e_{AC}$, $e_{CA}$, $e_{BD}$ and $e_{DB}$) in descending order of its weight. Starting from $e_{AC}$, the triangles ABC and ACD are examined, but neither of them fulfills the conditions for its marking. Then, the algorithm continues with $e_{BD}$. The triangles ABD and BCD are examined, and ABD is selected, thus $e_{BD}$ is marked and at the same time $e_{BA}$ and $e_{DA}$ are locked. Then, $e_{CA}$ is accessed and based on triangle ABC $e_{CA}$ is marked, while $e_{CB}$ and $e_{AB}$ are locked. Then, $e_{DB}$ is checked but is not marked. The algorithm continues with accessing the edges whose connected cells have a common side, i.e., $e_{CB}$ and $e_{BA}$. These edges are skipped, as they are locked. Then, $e_{CD}$ is examined and marked through the triangle BCD, and $e_{CB}$ and $e_{DB}$ are locked. The remaining edges are examined but cannot be marked. Figure 8b shows the result, which is an assignment that is both correct and duplicate-free.

For the special case where the marking of an edge can be done in two triangles, the selected triangle is the one whose locked edges from the potential marking have the largest sum of weight.

**Algorithm 2:** Point replication to cells.

**Input:** Coordinates of a point $o(x, y)$ and its type ($\tau$)
**Output:** A set with the ids of the assigned cells ($P$)

```
1  c_i ← getCell(x, y)
2  P ← {c_i.id}
3  if in no replication area then
4      return P
5  else if in merged duplicate-prone area of quartet q then
6      res ← MeDuPAr(graphDict.get(q), o, τ, c_i)
7      P.append(res)
8      res ← SupAr(graphDict.get(q'), o, τ, c_i)
9      P.append(res)
10     res ← SupAr(graphDict.get(q''), o, τ, c_i)
11     P.append(res)
12 else
13     s ← graphDict.get(q)
14     if s.e_ij.τ == o.τ then
15         P.append(c_j.id)
16     res ← SupAr(graphDict.get(q), o, τ, c_i)
17     P.append(res)
18     res ← SupAr(graphDict.get(q'), o, τ, c_i)
19     P.append(res)
20 return P
```

## 5.3 Algorithms for Duplicate-free Assignment

Algorithm 2 describes the process of point replication to cells. As input, it requires the coordinates of the point $o(x, y)$ and its type $\tau$. The output $P$ of the algorithm is a set of cell identifiers (ids) of the cells in which the point is assigned. Also, the cell $c_i$ that encloses $o$ is located. A set named $P$ is initialized with the cell id of $c_i$. Neighboring cell ids may be added to $P$. Algorithms 3 and 4 are invoked, named *MeDuPAr* and *SupAr* respectively. MeDuPAr constitutes the procedure of point assignment to cells given that it is located in the merged duplicate-prone area of a quartet. SupAr constitutes the procedure of point assignment located in a supplementary area of a quartet. Then, Algorithm 2 detects the exact area within cell $c_i$ in which the point is located. We identify the following three areas:

**(1) No replication area (line 3).** This area is depicted with white color in Figure 9. If the point is located in this area, it is not replicated to any other cell.

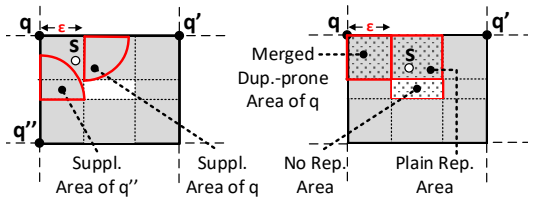



**Figure 9: Cell areas considered for point replication.**

**(2) Merged duplicate-prone area of a quartet (line 5).** This area constitutes a part of the merged duplicate-prone area of a quartet with reference point $q$. Simultaneously, a point located in this area, may be located in a supplementary area of other two quartets as well, with reference points $q'$ and $q''$. In a cell, there may exist at most four merged duplicate-prone areas (squared-shaped), each one belonging to a distinct quartet of cells. These areas are depicted with grey color as dark-shaded in Figure 9. If

**Algorithm 3:** MeDuPAr.

**Input:** Subgraph (*sub*), point $o$ $(x, y)$, point type ($\tau$) and cell ($c_i$)
**Output:** A set with the ids of the assigned cells ($P$)

```
1  P ← ∅
2  for c_j in c_1, c_2 cells do
       /* c_j: an adjacent cell to c_i                          */
3      if sub.e_ij.τ == τ and !sub.e_ij.isMarked() then
4          P.append(c_j.id)
5  if sub.e_il.τ == τ and !sub.e_il.isMarked() then
       /* c_l: cell with the common touching point with c_i
          */
6      if d(sub.ref, o) < ϵ then
7          P.append(c_l.id)
8      else
9          for c_j in c_1, c_2 cells do
10             if sub.e_ij.τ == τ and sub.e_ij.isMarked() then
                   /* c_l: cell with the common touching
                      point with c_i                           */
11                 P.append(c_l.id)
12 return P
```

**Algorithm 4:** SupAr.

**Input:** Subgraph (*sub*), point $o$ $(x, y)$, point type ($\tau$) and cell ($c_i$)
**Output:** A set with the id of the assigned cell ($P$)

```
1  P ← ∅
2  for c_j in c_1, c_2 cells do
       /* c_j: an adjacent cell to c_i                          */
3      if d(sub.ref, o) ≤ 2 * ϵ and d(c_j, o) ≤ ϵ then
4          if sub.e_ji.τ != τ and sub.e_ji.isMarked() then
5              if sub.e_ik.τ == τ and !sub.e_ik.isMarked() and
                  sub.e_jk.τ != τ and !sub.e_jk.isMarked() then
6                  P.append(c_k.id)
7              else if sub.e_il.τ == τ and !sub.e_il.isMarked() and
                  sub.e_jl.τ != τ and !sub.e_jl.isMarked() then
                   /* c_l: cell with the common touching
                      point with c_i                           */
8                  P.append(c_l.id)
9  return P
```

the point is located in the merged duplicate-prone area of the quartet with reference point $q$, then Algorithm 3 (MeDuPAr) is executed. Subsequently, Algorithm 4 (SupAr) is executed for the other two quartets with reference point $q'$ and $q''$, which are the nearest to the $q$ reference point. This algorithm checks if the point is located to a supplementary area of a triad in the respective quartet. Both Algorithms 3 and 4 require as arguments the quartet's subgraph *sub*, the point $o$, its type ($\tau$) and its native cell ($c_i$). Concerning the subgraph's argument, this is fetched from the dictionary that handles the graphs (*graphDict*) by providing the quartet's reference point. Also, the two algorithms output a set with the ids of the assigned cells in the quartet, which are appended to $P$ set.

**(3) Plain replication area (line 12).** A point located in this area, is candidate for replication at a cell ($c_k$) as $MINDIST(o, c_k) \leq \epsilon$. This is captured in line 12, when the point does not exist in the two aforementioned types of areas. In a cell, these areas are four in total, depicted with grey color in Figure 9. If the agreement between the point's native cell and its adjacent cell ($c_k$) is the same with the point's type, then the id of the adjacent cell is appended to $P$ set (line 14). For this, the subgraph of the quartet with the nearest reference point $q$ to the point $o$ (line 13) is fetched from the *graphDict* dictionary at first. Then, the type ($\tau$) of the $e_{ij}$ edge of the subgraph is checked. Finally, Algorithm 4 follows for the quartets $q$ and $q'$, whose reference points are the nearest to the point, as it may be located in a supplementary area of them.

Both Algorithms 3 and 4 initialize an empty set ($P$), which will be their output. The set is appended with the id of the cells in which the point is replicated.

Algorithm 3 starts by checking the replication procedure of the point to cells $c_1$ and $c_2$ (stated as $c_j$ in the for loop in line 2) that have common borders with the point's native cell ($c_i$) (line 3). For this, the type of the $e_{ij}$ edge type checked and its state. The Algorithm is completed with the checking procedure for replicating the point to the cell ($c_l$) that has only one common touching point with the point's native cell (line 5). The point will be replicated to $c_l$ if its distance is less than $\epsilon$ to the quartet's reference point or if one of the edges that connects the native cell ($c_i$) with its adjacent cells ($c_j$ in the for loop) is marked and has the same type with the point.

Algorithm 4 considers at first the two cells ($c_1$ and $c_2$, stated as $c_j$ in the for loop in line 2) that share common borders with point's native cell ($c_i$). The point is checked for being located in a potential supplementary area defined by the $c_j$ cell. This is validated by calculating the distance between the quartet's reference point and the point, and the distance between the $c_j$ cell and the point (line 3). If the respective conditions are both fulfilled, then the existence of the supplementary area for the point's type is checked. This takes place by validating the type and the state of the edge that indicates the replication of points from $c_j$ to $c_i$ cell ($e_{ji}$). If the type that edge is opposite of the point's type and is marked, then the supplementary area exists for the point (line 4). The algorithm continues with the assignment of the point to one of the other two cells in the quartet ($c_k$ or $c_l$). The point is assigned to the cell whose connecting edges with the point's native cell ($c_i$) and the cell from which supplementary area is defined ($c_j$), fulfill specific conditions that concern to their type and state (lines 5 and 7).

## 6 PARALLEL DISTANCE JOIN ALGORITHM

In this section, we present our algorithm for parallel $\epsilon$-distance spatial joins, which is designed and implemented in Apache Spark.

### 6.1 Parallel Distance Join Execution

Algorithm 5 describes $\epsilon$-distance join in the Spark context. The commands in bold correspond to Spark operations. The algorithm can be split in discrete steps. Initially, the *grid* is determined given the minimum bounding box rectangle of data ($m$) and the distance join threshold ($\epsilon$), in line 1.

**Sampling and Agreement-based Grid Construction**. This step includes the loading of $R$ and $S$ data sets from HDFS in raw format (txt files) into two RDDs that handle the tuples of each set. For this purpose the *map* function is used (lines 2 and 3). Then, data sampling on each RDD follows, and the sample is fetched to the driver. The two data samples are exploited to supply with information the grid cells (lines 4 and 5), required

**Algorithm 5:** $\epsilon$-distance Spatial Join Algorithm in Spark.

**Input:** MBR of data space ($m$), dist. threshold ($\epsilon$), $R$ set ($pathR$), $S$ set ($pathS$), SparkContext ($sc$), sample size ($\phi$)
**Output:** Pairs ($p$) of matched points ($\{r, s\}$)

```
1  grid ← Grid(m, ε)
2  rddR ← sc.textFile(pathR).map(line → tup)
3  rddS ← sc.textFile(pathS).map(line → tup)
4  rddR.sample(φ).forEach(tup → grid.addR(tup.x, tup.y))
5  rddS.sample(φ).forEach(tup → grid.addS(tup.x, tup.y))
6  gBr ← sc.broadcast(grid)
7  pairRddR ← rddR.flatMapToPair(t → tList(gBr.getIds(o, R)))
8  pairRddS ← rddS.flatMapToPair(t → tList(gBr.getIds(o, S)))
9  p ← pairRddR.join(pairRddS).filter(d(r_i, s_j) ≤ ε)
10 return p
```

for the formation of the agreements (they do not take place yet). Then, the grid is broadcast to every node of the cluster (line 6).

**Spatial Mapping of Points**. The mapping procedure is based on the broadcast grid, as it maps every point to a set of 1D values based on its coordinates $(x, y)$. In this phase, agreements are formed while requesting the mapping values of points. Essentially, this is the point where the algorithms of Section 5 are used. The point assignment to 1D values is combined with the mapping of each RDD to a PairRDD, which is practically an RDD of key-value pairs. This is achieved by means of Spark's transformation $flatMapToPair$. In our case, the key is set as a single 1D value, while the value constitutes the tuple. Thus, a single RDD tuple may be matched to more than one PairRDD entries, particularly as many times as the number of assigned 1D values (lines 7 and 8).

**Assigning and Joining the Partitions.** Having the two PairRDDs formed, data shuffling occurs, ensuring that the PairRDD entries with the same key will be located on the same node. After shuffling, the data on each node are hash-joined locally on the key value. This determines the set of candidate join results (line 9).

**Computing distance join at partition-level.** Directly after the production of a candidate pair of points, their actual distance is computed. If $d(r, s) \leq \epsilon$, then pair is reported in the final result set, otherwise it is disregarded. This comprises the *refinement* phase, where false positives are discarded (also shown in line 9).

## 6.2 Assignment of Cells to Workers

Based on the sample, each cell can be associated with an estimate of the induced cost for processing. This cost is practically the product of the estimated points of sets $R$ and $S$ that will eventually be in the cell. The remaining question is how to find an assignment of the set of $C$ cells to $W$ workers in Spark that achieves load balancing. We set as optimization criterion that the maximum number of join results per worker should be minimized. In turn, this is expected to minimize the maximum processing time per worker.

This problem is equivalent to the *Multiprocessor Scheduling Problem* [11], which is known to be NP-hard. Therefore, we utilize a greedy algorithm called *LPT (Longest Processing Time)* which sorts the cells' costs based on number of join records and then assigns them to the worker with the lowest aggregate number of join results so far. Notice that the reason that makes the LPT algorithm applicable in our setting is that we have an estimate of the number of join records per cell, due to the sampling procedure.

| Product | Codename | Cardinality |
|---|---|---|
| TIGER/Area Hydrography | $R_1$ | 94.1M |
| OSM/Parks | $R_2$ | 42.7M |
| SYNTHETIC/Gaussian | $S_1$ | 100M |
| SYNTHETIC/Gaussian | $S_2$ | 100M |

**Table 2: Overview of the data sets used in the experiments.**

# 7 EXPERIMENTS

In this section, we present the results of the experimental evaluation. Our code[1] is written in Java and Apache Spark version v3.1.3.

## 7.1 Experimental Setup

**Data sets.** In our evaluation we use real data sets from TIGER[2] and OpenSteetMap[3], summarized in Table 2, and obtained from the SpatialHadoop project[4]. To test scalability with larger data sets, we also use synthetic data sets that follow the Gaussian distribution consist of 30 clustered areas of points with standard deviation in the range $[0.1, 0.8]$. They were generated in the same minimum bounding rectangle of the real data sets. In all experiments we use as sample the 3% of each data set. We found that this sample size offers the best performance in our experiments.

**Algorithms.** We study the performance of our algorithm presented in Section 6 which uses adaptive replication. We use two variants of this algorithm, denoted LPiB and DIFF, which differ on how to instantiate the graph of agreement, as outlined in Sect. 4.3.

We compare against three adaptations of PBSM for Apache Spark that also rely on grid partitioning and use a hash-based partitioner to distribute partitions to workers. Specifically, the two variants of PBSM, denoted as UNI(R) and UNI(S), replicate the data sets $R$ and $S$ respectively, having the same grid resolution ($2\epsilon \times 2\epsilon$) as LPiB and DIFF. The third variant of PBSM, denoted as $\epsilon$-grid, has $\epsilon \times \epsilon$ grid resolution and replicates the data set with the fewest objects. The choice of PBSM is justified as it has been indicated as one of the best performing algorithms for in-memory spatial joins [12], it has been evaluated for parallel spatial joins [21], and it is used in most distributed processing systems for big spatial data [1, 6, 25, 27].

Additionally, we employ the Apache Sedona v1.4.1 processing framework, denoted simply as Sedona. The join execution in Sedona is performed in three phases that pertain to partitioning, indexing and join computation. At first, the objects of the data sets are assigned to partitions. We choose the QuadTree spatial partitioning scheme. The structure is built on the driver regarding one of the data sets by fetching a sample. For this, we select the set with the fewest objects as on this set replication will be induced (objects may be assigned to more than one partitions). After the data partitioning and assignment to the executors, spatial indexing takes place. A local RTree index is built (per partition) on the set that contains the most points. Then, join is computed by issuing queries from the other set to the index. We opt for the usage index, as Sedona performs better than not using index.

**Metrics.** We use the following metrics: (a) execution time of the job that performs the spatial distance join, (b) number of replicated data objects, and (c) the size of the shuffled data over

[1] https://github.com/nkoutroumanis/parallel-spatial-joins
[2] https://www.census.gov/programs-surveys/geography/guidance/tiger-data-products-guide.html
[3] https://www.openstreetmap.org
[4] http://spatialhadoop.cs.umn.edu/datasets.html

| | |
|---|---|
| **Distance threshold $\epsilon$** | 0.009, **0.012**, 0.015, 0.018 |
| **Covering surface (%)** | $4{\cdot}10^{-4}$, $\mathbf{6{\cdot}10^{-4}}$, $8{\cdot}10^{-4}$, $9{\cdot}10^{-4}$ |
| **Data size** ($S_1 \bowtie S_2$) | **x1**, x2, x4, x6, x8 |
| **Number of nodes** | 4, 6, 8, 10, **12** |
| **Tuple size factor** | $\mathbf{f_0}$, $f_1$, $f_2$, $f_3$, $f_4$ |
| **Data sets combinations** | $S_1 \bowtie S_2$, $R_1 \bowtie S_1$, $R_2 \bowtie R_1$ |

**Table 3: Experimental parameters (default values in bold).**

the network. In all experiments, we report the average results of 10 executions.

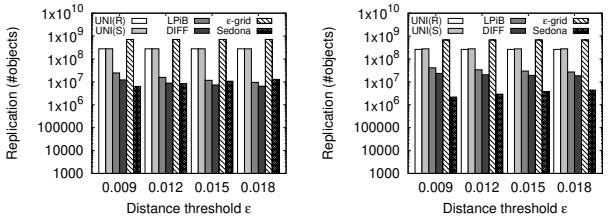
**Platform.** The experiments are performed in the Okeanos-Knossos IaaS platform, a cloud service that offers virtual computing and storage services, supported by GRNET (https://grnet.gr) for research purposes. In total, 15 VMs are used, running Ubuntu 16.04.6 LTS, each equipped with 4 CPU cores, 8GB RAM and 30GB system disk. Twelve of these VMs have a mounted disk of 102GB. The offered attachable disk of Okeanos-Knossos platform is based on the Ceph (https://ceph.io/) storage system, which is a distributed block level storage. The VMs with the attached mounted disk run the HDFS v3.2.1 and function as Datanodes and NodeManagers. The remaining 3 VMs act as NameNode, ResourceManager (YARN) and Driver for the running jobs. The HDFS uses the default configuration (128MB block size and replication factor of 3).

**Parameters.** Table 3 summarizes the experimental parameters: distance threshold $\epsilon$, data size, the number of nodes (equal to the number of executors) used for the join, and tuple size factor (i.e., the overhead in bytes of each tuple, besides the spatial information). The latter is used because real-world data often contain extra data apart from spatial coordinates, such as names, descriptions, etc., that need to be transferred, which (i) poses an overhead in distributed processing, and (ii) cannot be efficiently handled in a post-processing step as in centralized settings, because retrieving this information from a distributed data set has non-negligible cost.

The number of Spark partitions for the join operation on the PairRDDs is set to 96 as default. In the experiments where we vary the data set size, we increase it to 192 for 400M and by 96 for each subsequent data size factor. Also, for the experiments where we vary the tuple size, the number of partitions is set to 192 except for the combination of the real data sets ($R_1 \bowtie R_2$) which is set to 120. For every case, the same number of partitions is applied in all algorithms. Also notice that our largest synthetic data set contains (800M) of data objects.

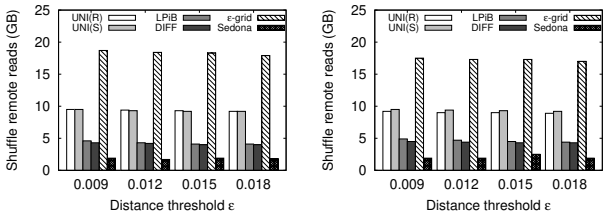
## 7.2 Experimental Results

*7.2.1 Effect of varying the distance threshold $\epsilon$.* Figures 10, 11 and 12 show the effect of varying radius on replication, shuffled remote reads and execution time for performance of spatial join for two combinations of data sets (synthetic and real with synthetic). In Figures 10a and 10b, the number of replicated data is shown in log scale. Our algorithms (LPiB and DIFF) outperform the UNI(R), UNI(S) and $\epsilon$-grid by at least one order of magnitude for all values of $\epsilon$, verifying our motivation for minimizing replication. When the distance threshold is increased, we notice that our algorithms require less replication. This is attributed to the fact that for larger $\epsilon$, the grid cells also become larger in size, thus less replication is induced as in both combinations, the data is skewed. The $\epsilon$-grid algorithm has the highest replication (7.1x), while the replication in Sedona for the synthetic data sets is in the same level as in LPiB/DIFF due to the fact that the resulted



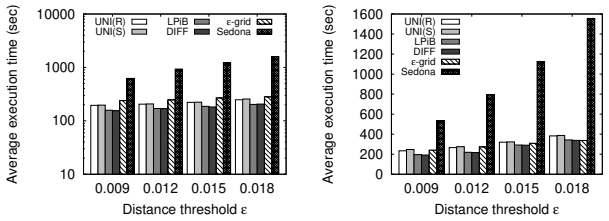

**(a) $S_1 \bowtie S_2$** **(b) $R_1 \bowtie S_1$**

**Figure 10: Effect of varying radius on replication.**




**(a) $S_1 \bowtie S_2$** **(b) $R_1 \bowtie S_1$**

**Figure 11: Effect of varying radius on shuffle remote reads.**




**(a) $S_1 \bowtie S_2$** **(b) $R_1 \bowtie S_1$**

**Figure 12: Effect of varying radius on execution time.**

partitions from the QuadTree space partitioning are quite large. However, as will be shown below, this has a negative impact on performance. Complementary to this result, Table 4 reports the join selectivity.

In Figures 11a and 11b, the size of data due to shuffled remote reads is shown. In all cases, our algorithms require much less data to be read/transferred over the network than UNI(R), UNI(S) and $\epsilon$-grid. Sedona has the lowest shuffled remote data remote reads.

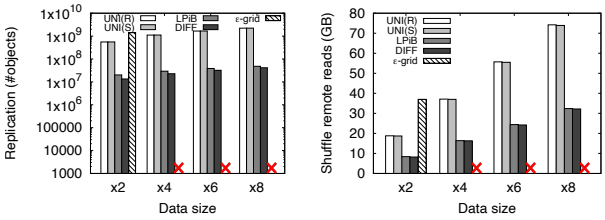
Figures 12a and 12b show the execution time for the join for the two data set combinations. In all cases, when $\epsilon$ is increased, more time is required as the output size also increases. For the synthetic sets, LPiB and DIFF outperform the competitors. The difference in the performance of the best case of the competitors and the proposed approaches is on average 18.6%. The respective difference for the combination of the real with the synthetic sets is 10.7%.

Notice that in all experiments Sedona requires almost one order of magnitude more time compared to our algorithm. This is the result of using large partitions that may reduce replication, but increase the cost of join processing, thus having huge negative impact on execution time.

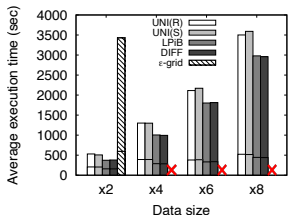
*7.2.2 Scalability with the size of data.* In Figure 13, we study the scalability of all algorithms for increased sizes of data. In all

| $S_1 \bowtie S_2$ | | | | |
|---|---|---|---|---|
| **Distance threshold $\epsilon$** | 0.009 | 0.012 | 0.015 | 0.018 |
| **Selectivity (%)** | $5.5{\cdot}10^{-6}$ | $9.8{\cdot}10^{-6}$ | $1.5{\cdot}10^{-5}$ | $2.2{\cdot}10^{-5}$ |
| **Join results** | 553M | 984M | 1.5B | 2.2B |
| **Data size** | x2 | x4 | x6 | x8 |
| **Selectivity (%)** | $9.8{\cdot}10^{-6}$ | $9.8{\cdot}10^{-6}$ | $9.8{\cdot}10^{-6}$ | $9.8{\cdot}10^{-6}$ |
| **Join results** | 3.9B | 15.8B | 35.5B | 63.1B |
| $R_1 \bowtie S_1$ | | | | |
| **Distance threshold $\epsilon$** | 0.009 | 0.012 | 0.015 | 0.018 |
| **Selectivity (%)** | $1.3{\cdot}10^{-5}$ | $2.4{\cdot}10^{-5}$ | $3.8{\cdot}10^{-5}$ | $5.5{\cdot}10^{-5}$ |
| **Join results** | 1.3B | 2.3B | 3.6B | 5.2B |
| $R_1 \bowtie R_2$, **Selectivity (%):** $5.7{\cdot}10^{-5}$, **Join results:** 2.3B | | | | |

**Table 4: Result set selectivity and join results.**




**(a) Replication** **(b) Shuffle remote reads**

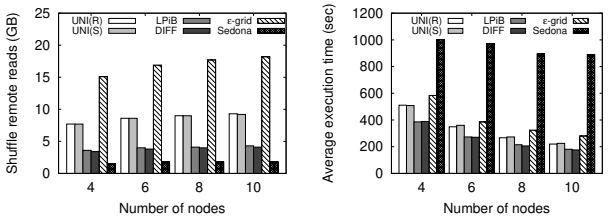



**(c) Execution time**

**Figure 13: Effect of varying data set size for $S_1 \bowtie S_2$.**

cases, the performance gain of our algorithms is sustained, even for larger data sets. It is noteworthy that the replicated objects for LPiB/DIFF methods against the baselines, remain low with the variation of the data size (Figure 13a). Also, the shuffled data increases with a much smaller rate for our algorithms compared to UNI(R) and UNI(S) (Figure 13b). This shows that our algorithms can save more remote reads than the competitors for larger data sets. In terms of execution time (Figure 13c), the difference increases with larger data sets, showing that our algorithms are more scalable than the competitors. The difference between the performance of the best case of the baseline approaches and the proposed approaches is 19.9%. respectively. Notice that the red 'x' in the figures indicates that $\epsilon$-grid did not finish its execution due to an out of memory error, as a result of the high replication.

*7.2.3 Construction time.* Also, in Figure 13c, we use stacked bars to split the execution time between construction (lower part of the stacked bar) and join processing (upper part). The construction time includes the time spent for our Algorithms 2, 3 and 4, as well as the time for data shuffling. Figure 13c shows that (a) as we increase that size of the input data, most of the execution time is spent on join processing, and (b) the construction time is only slightly increased, although the cost of shuffling is included. Therefore, the phase of construction is very efficient for both LPiB and DIFF.




**(a) Shuffle remote reads** **(b) Execution time**

**Figure 14: Effect of varying the number of nodes ($S_1 \bowtie S_2$).**

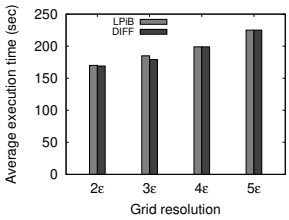


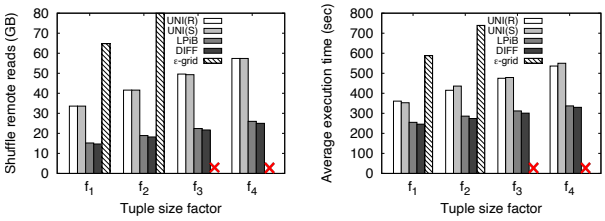

**Figure 15: Effect of varying the grid resolution ($S_1 \bowtie S_2$).**

*7.2.4 Scalability with the number of nodes.* Figure 14 shows the performance on the synthetic data sets when varying the number of nodes in the cluster. All algorithms perform better with more executors, showing reduced execution time and a slight increase on the shuffle remote reads. Note that the percentage difference in execution time is larger when nodes are already few, i.e., when 4 nodes are increased to 6, the drop percentage is 30% on average, while when 8 nodes are increased to 10, the respective percentage is between 13.5% and 17.5%.

*7.2.5 Effect of varying the grid resolution.* Figure 15 shows the effect of varying the grid resolution from $2\epsilon$ (fine-grained) to $5\epsilon$ (coarse-grained) on the performance of our algorithm variants, LPiB and DIFF, for the synthetic sets ($S_1 \bowtie S_2$). The results show that as we increase the cell size, the average execution time increases too. When the spatial extent of a cell is increased, a cell contains more objects and this imposes extra cost per cell for the join computation. This justifies the use of $2\epsilon$ as grid resolution, for both LPiB and DIFF, as this setting achieves the best performance.

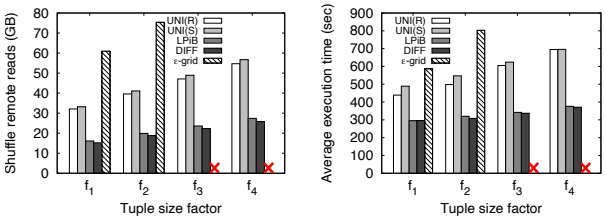
*7.2.6 Varying the tuple size.* Figures 16, 17 and 18 show the effect of increasing the tuple size. In practice, spatial data sets also have additional attributes besides the location information. Thus, it is interesting to study the performance of join algorithms for larger tuple sizes. In synthetic sets (Figures 16a and 16b), in the combination of synthetic and real set (Figures 17a and 17b) and in real sets (Figures 18a and 18b), the tuple size has a significant impact on the execution time. Specifically, the addition of extra fields worsens the performance for the competitor methods. This is because more objects are replicated than in our algorithms, and the extra attributes increase the cost of shuffling. Instead, the LPiB and DIFF approaches remain at the same levels with slight difference in shuffle remote reads and average execution time for the different tuple size factors. In all cases, the $\epsilon$-grid scores the greatest shuffle remote reads due to high replication that reflects on its performance. Also, the join processing time of each algorithm increases, because larger objects are processed.

The average percentage differences for each tuple size factor among the best case competitor approach and the LPiB/DIFF



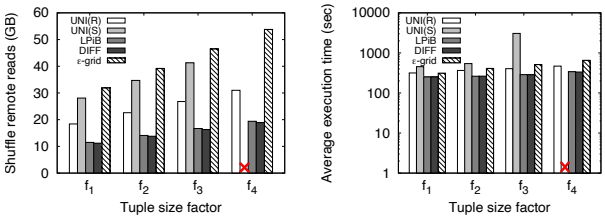

(a) Shuffle remote reads (b) Execution time

**Figure 16: Effect of increasing tuple size ($S_1 \bowtie S_2$).**




(a) Shuffle remote reads (b) Execution time

**Figure 17: Effect of increasing tuple size ($R_1 \bowtie S_1$).**




(a) Shuffle remote reads (b) Execution time

**Figure 18: Effect of increasing tuple size ($R_2 \bowtie R_1$).**

| Inclusion of extra attributes | On join | | On post-processing | |
|---|---|---|---|---|
| Method | LPiB | DIFF | LPiB | DIFF |
| Exec. time | 255 | 246 | 727 | 772 |

**Table 5: Execution time (in seconds) of LPiB and DIFF with two types of inclusion of extra attributes for $S_1 \bowtie S_2$.**

approaches are 34.8% for the synthetic sets, 40.4% for the combination of real and synthetic sets, and 26.5% for the real sets. This shows that the performance gain of our algorithms increases for data sets with many non-spatial attributes.

We also tested an alternative solution for including the extra attributes in the result set. Instead of including them to the tuples when computing the join, we employ a post-processing step that fetches and adds them to the result set. This is done by means of two joins on the id of the tuples, between the two data sets that contain the tuples with the extra attributes (R and S) and the result pairs of the spatial join. Table 5 shows the execution time for the default setup for the $f_1$ factor. We observe that the gain in performance is almost 3x greater when the attributes are already included.

*7.2.7 Duplicate-free and non duplicate-free assignment with deduplication step.* Table 6 shows the average execution time for two variations of the LPiB and DIFF methods in the default experimental setup. The two variations differ on the fact that the one proceeds with duplicate-free assignment, while the other uses a simplified algorithm that does produce duplicates. For the latter, a deduplication step is incorporated as a `distinct` operator, after the computation of the join. The `distinct` operator is performed in parallel across the cluster, since collecting the data to the driver is infeasible for really large outputs (in this setup, 985M objects). We observe that applying the deduplication at the end is much more costly (more than 7x) than using the duplicate-free approach.

| Assignment | Duplicate-free | | Non duplicate-free (with deduplication) | |
|---|---|---|---|---|
| Method | LPiB | DIFF | LPiB | DIFF |
| Exec. Time | 170 | 169 | 1224 | 1245 |

**Table 6: Execution time (in seconds) of LPiB and DIFF with duplicate-free and non duplicate-free assignment ($S_1 \bowtie S_2$).**

| Cell assignment to workers | Hash-based | | LPT | |
|---|---|---|---|---|
| Method | LPiB | DIFF | LPiB | DIFF |
| Exec. time $(S_1 \bowtie S_2)$x4 | 1056 | 1044 | 1005 | 995 |
| Exec. time $(R_2 \bowtie R_1)$ | 212 | 211 | 198 | 203 |

**Table 7: Execution time (in seconds) of LPiB and DIFF with hash and LPT assignment for two combinations of data sets.**

*7.2.8 Effect of load balancing.* Table 7 shows effect of the LPT load balancing mechanism on the execution time of LPiB and DIFF. For this purpose, we disable the use of LPT, thus the alternative configuration uses Spark's default hash-based assignment of partitions (cells) to workers. We observe that LPT achieves slightly better performance. For the synthetic data sets the average gain in performance is 4.75%, while for the combination of the synthetic with the real set, it is 5.2%. Notice that this largely depends on the spatial skewness of input data. Essentially, when the distribution of pairs of objects in cells is highly skewed, LPT can perform the allocation of cells to workers in a more fair way, thus resulting in more effective parallelization and improved execution time.

## 8 CONCLUSIONS AND FUTURE WORK

In this paper we propose an approach for adaptive replication of objects among workload partitions in order to process efficiently spatial distance joins in parallel. Our work is based on a graph-based framework that allows to keep track of local decisions about replication, and leads to a correct and duplicate-free results. The approach presents a substantial gain in performance when compared to the PBSM algorithm variations and to Apache Sedona, in which the replication occurs exclusively in one of the data sets. In our future work, we plan to extend the abstraction of the graph of agreements for other spatial objects, such as polygons and polylines. We also intend to generalize our graph-based abstraction for other partitioning schemes, such as QuadTrees. Finally, deriving a theoretical cost model for our algorithms is of interest.

## ACKNOWLEDGEMENTS

This work was supported by the Horizon Europe R&I programme EMERALDS under the GA No. 101093051, and by the CHOROLOGOS research project, funded by the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under GA No. HFRI-FM17-81.

# REFERENCES


[1] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. 2013. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *Proc. VLDB Endow.* 6, 11 (2013), 1009–1020.

[2] Panagiotis Bouros and Nikos Mamoulis. 2019. Spatial joins: what's next? *ACM SIGSPATIAL Special* 11, 1 (2019), 13–21.

[3] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1996. Parallel Processing of Spatial Joins Using R-trees. In *Proc. of ICDE*, Stanley Y. W. Su (Ed.). IEEE Computer Society, 258–265.

[4] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of OSDI*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 137–150.

[5] Jens-Peter Dittrich and Bernhard Seeger. 2000. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *Proc. of ICDE*, David B. Lomet and Gerhard Weikum (Eds.). 535–546.

[6] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *Proc. of ICDE*, Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman (Eds.). 1352–1363.

[7] Ahmed Eldawy and Mohamed F. Mokbel. 2016. The Era of Big Spatial Data: A Survey. *Found. Trends Databases* 6, 3-4 (2016), 163–273.

[8] Francisco García-García, Antonio Corral, Luis Iribarne, and Michael Vassilakopoulos. 2020. Improving Distance-Join Query processing with Voronoi-Diagram based partitioning in SpatialHadoop. *Future Gener. Comput. Syst.* 111 (2020), 723–740.

[9] Francisco García-García, Antonio Corral, Luis Iribarne, and Michael Vassilakopoulos. 2023. Efficient distributed algorithms for distance join queries in spark-based spatial analytics systems. *Int. J. Gen. Syst.* 52, 3 (2023), 206–250.

[10] Francisco García-García, Antonio Corral, Luis Iribarne, Michael Vassilakopoulos, and Yannis Manolopoulos. 2018. Efficient large-scale distance-based join queries in SpatialHadoop. *GeoInformatica* 22, 2 (2018), 171–209.

[11] M. R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman.

[12] Sadegh Nobari, Qiang Qu, and Christian S. Jensen. 2017. In-Memory Spatial Join: The Data Matters!. In *Proc. of EDBT*, Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß (Eds.). 462–465.

[13] Jignesh M. Patel and David J. DeWitt. 1996. Partition Based Spatial-Merge Join. In *Proc. of SIGMOD*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 259–270.

[14] Jignesh M. Patel and David J. DeWitt. 2000. Clone join and shadow join: two parallel spatial join algorithms. In *Proc. of GIS*, Ki-Joune Li, Kia Makki, Niki Pissinou, and Siva Ravada (Eds.). 54–61.

[15] Mirjana Pavlovic, Thomas Heinis, Farhan Tauheed, Panagiotis Karras, and Anastasia Ailamaki. 2016. TRANSFORMERS: Robust spatial joins on non-uniform data distributions. In *Proc. of ICDE*. 673–684.

[16] Tilemachos Pechlivanoglou, Mahmoud Alsaeed, and Manos Papagelis. 2020. MRSweep: Distributed In-Memory Sweep-line for Scalable Object Intersection Problems. In *Proc. of DSAA*, Geoffrey I. Webb, Zhongfei Zhang, Vincent S. Tseng, Graham Williams, Michalis Vlachos, and Longbing Cao (Eds.). IEEE, 324–333.

[17] Suprio Ray, Bogdan Simion, Angela Demke Brown, and Ryan Johnson. 2014. Skew-resistant parallel in-memory spatial join. In *Proc. of SSDBM*, Christian S. Jensen, Hua Lu, Torben Bach Pedersen, Christian Thomsen, and Kristian Torp (Eds.). 6:1–6:12.

[18] Thomas Seidl, Sergej Fries, and Brigitte Boden. 2013. MR-DSJ: Distance-Based Self-Join for Large-Scale Vector Data Analysis with MapReduce. In *Proc. of BTW*. 37–56.

[19] Darius Sidlauskas and Christian S. Jensen. 2014. Spatial Joins in Main Memory: Implementation Matters! *Proc. VLDB Endow.* 8, 1 (2014), 97–100.

[20] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. *Proc. VLDB Endow.* 9, 13 (2016), 1565–1568.

[21] Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2019. Parallel In-Memory Evaluation of Spatial Joins. In *Proc. of SIGSPATIAL*, Farnoush Banaei Kashani, Goce Trajcevski, Ralf Hartmut Güting, Lars Kulik, and Shawn D. Newsam (Eds.). 516–519.

[22] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *Proc. of SIGMOD*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1071–1085.

[23] Jie Yang and Satish Puri. 2020. Efficient Parallel and Adaptive Partitioning for Load-balancing in Spatial Join. In *Proc. of IPDPS*. 810–820.

[24] Simin You, Jianting Zhang, and Le Gruenwald. 2015. Large-scale spatial join query processing in Cloud. In *Proc. of ICDE Workshops*. 34–41.

[25] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. GeoSpark: a cluster computing framework for processing large-scale spatial data. In *Proc. of SIGSPATIAL*, Jie Bao, Christian Sengstock, Mohammed Eunus Ali, Yan Huang, Michael Gertz, Matthias Renz, and Jagan Sankaranarayanan (Eds.). 70:1–70:4.

[26] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proc. of HotCloud*, Erich M. Nahum and Dongyan Xu (Eds.). USENIX Association.

[27] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. 2009. SJMR: Parallelizing spatial join with MapReduce on clusters. In *Proc. of Cluster Computing*. 1–8.

[28] Xiaofang Zhou, David J. Abel, and David Truffet. 1997. Data Partitioning for Parallel Spatial Join Processing. In *Proc. of SSD*, Michel Scholl and Agnès Voisard (Eds.). 178–196.