A Balanced Solution for the Partition-based Spatial Merge Join in MapReduce

Sara Migliorini
Dept. of Computer Science, University of Verona
sara.migliorini@univr.it

Alberto Belussi
Dept. of Computer Science, University of Verona
alberto.belussi@univr.it

ABSTRACT

Several MapReduce frameworks have been developed in recent years in order to cope with the need to process an increasing amount of data. Moreover, some extensions of them have been proposed to deal with particular kind of information, like the spatial one. In this paper we will refer to SpatialHadoop, a spatial extension of Apache Hadoop which provides a rich set of spatial data types and operations. In the geo-spatial domain, spatial join is considered a fundamental operation for performing data analysis. However, the join operation is generally classified as a critical task to be performed in MapReduce, since it requires to process two datasets at time. Several different solutions have been proposed in literature for efficiently performing a spatial join which may or may not require the presence of a spatial index computed on both datasets or only one of them. As already discussed in literature, the efficiency of such operation depends on the ability to both prune unnecessary data as soon as possible and to provide a balanced amount of work to be done by each parallelly executed task.

In this paper, we take a step forward in this direction by proposing an evolution of the Partition-based Spatial Merge Join algorithm which tries to completely exploit the benefit of the parallelism induced by the MapReduce framework. In particular, we concentrate on the partition phase which has to produce filtered balanced and meaningful subdivisions of the original datasets.

KEYWORDS

Big Data, Partitioning, Spatial join, Balanced tasks, MapReduce

1 INTRODUCTION

The MapReduce paradigm has been specifically developed for processing huge amount of data in an efficient way. In particular, it requires to subdivide the desired analysis operation into two subsequent phases: the first one is called *map* and it performs in parallel the same operation on independent chunk of data, producing a set of partial results. These partial results are (possibly) combined by the second phase which is called *reduce* and may be parallelized as well. Nowadays there are several frameworks that implement the MapReduce paradigm, undoubtedly the most famous ones are Apache Hadoop [15] and Apache Spark [17]. Moreover, in order to cope with the specific needs of particular kinds of information, such as the spatial and temporal one, some extensions have been developed which provides the implementation of the necessary data types and operations. As regards to the spatial context, notable systems are SpatialHadoop [8] and GeoSpark [16].

The basic partitioning approach traditionally implemented by a MapReduce framework essentially uses a predefined amount of bytes as splitting criteria, without considering the content of such data. In other words, given the original dataset, its records are placed inside the current partition (or split) until a given size threshold is reached, then another partition is initialized, and so on. Such an approach can be suitable in the general case when all the dataset content has to be processed. However, in other cases when data are analyzed using selective queries based on some attributes, such as time intervals or spatial regions, this approach may be very inefficient, since the correlation between data instances is completely ignored. For this reason, in the spatial context, some global indexing (or partitioning) techniques have been developed, which try to place inside the same partition data which are spatially correlated in some way (i.e., placing inside the same partition nearby objects).

In geo-spatial applications, spatial join is considered an essential operation for data analysis [4, 5], since it allows the data analyst to discover important correlations between data or enrich the available information. A spatial join is a multi-dimensional join specifically tailored for spatial data, in particular given two datasets A and B each one characterized by a geometric attribute, called g and f respectively, a spatial join $A \bowtie B$ returns the set of pairs (a, b) such that $a \in A$ and $b \in B$ and the geometric intersection between the attributes $a.f$ and $b.g$ is not empty. For instance, we can consider the case in which a geographer needs to compute the spatial join between two huge datasets, one containing the main roads and one the water areas, in order to predict the main roads of her country which could be subject to flooding risks. Similarly, a geographer may need to perform a join between a dataset containing the main roads and another one representing the administrative subdivisions of her country, in order to study the density of the road network in each state.

However the join operation has been traditionally considered a critical operation in MapReduce for two main reasons: (1) the need to process two distinct datasets (files) at the same time, (2) the need to perform some pruning or filtering operation in order to reduce the amount of unnecessary comparisons. In order to solve such problems, many efforts have been devoted in recent years leading to the development of different MapReduce implementation of the join [6, 10] each one applicable to a particular context. This also holds in the spatial context, where several algorithms have been defined and implemented, which essentially differ for the use of a spatial index and in the way this index is built and used. At this regard SpatialHadoop is one of the available framework which provides an implementation for all these algorithms [9] that can also be combined with different kinds of indexes (partitioning techniques) [7]. In general, none of these algorithms can be considered better than the others, but the choice might depend on the characteristics of the involved datasets [2].

As we will describe in Sect. 2, the application of a spatial partitioning technique before the execution of a join operation is particularly useful in order to both discard entire partitions from the analysis, or to balance the amount of work to be done in

© 2020 Copyright for this paper by its author(s). Published in the Workshop Proceedings of the EDBT/ICDT 2020 Joint Conference (March 30-April 2, 2020, Copenhagen, Denmark) on CEUR-WS.org. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

CEUR-WS.org/Vol-2578/DARLIAP14.pdf

parallel (typically by each map task). Relatively to this aspect, in literature has been extensively studied the importance of balancing the amount of work to be done by each map task in order to completely exploit the parallelism induced by the MapReduce paradigm [1, 3]. However, the preliminary application of a spatial partitioning technique comes with its cost and sometimes it is justified only if such new organization of data can be reused several time, absorbing such initial cost. Moreover, the right partitioning technique to be applied may depends not only on the dataset characteristics, such as its distribution [3], but also from the type of analysis will be performed [12].

From these considerations, since the join operation requires to process two distinct datasets together, the choice of the right partitioning technique has necessarily to consider both datasets together in order to balance the amount of work to be done by each map task. As we will better explain in Sect. 3, given two datasets A and B, each one individually partitioned with the most appropriate partitioning technique, it may happen that the splits obtained by combining them for the join will not necessarily produce the overall better setting w.r.t. the balancing criterion.

In this paper we consider the implementation of the Partition Based Spatial Merge Join [13] provided by SpatialHadoop, denoted as SJMR, which is the only spatial join algorithm that does not rely on the preliminary application of individual spatial partition techniques on the input datasets, but it defines a partition grid based on both datasets together. Given such algorithm we enhance the definition of a common partitioning grid in order to both preliminary prune unnecessary data and promote the balancing of the obtained splits. Such partitioning technique has to consider the spatial characteristics of both dataset together, such as the spatial distribution of its objects as well as the global covered area.

2 BACKGROUND

SpatialHadoop provides four different spatial join algorithms which essentially differ for the use of spatial indexes or for the way data are repartitioned on the fly. In particular, the simplest solution is represented by the DJNI algorithm. DJNI stands for Distributed Join with No Index, it uses the default random partitioning technique, which is provided by any MapReduce framework and is based only on the size constraint. In other words, it does not involve any preliminary repartition of the data based on their spatial properties (no index or global repartitioning is applied). Since DJNI cannot rely on any spatial property, given two datasets D_i and D_j subdivided into n and m partitions, respectively, the number of map tasks to be instantiated is equal to the Cartesian product $n \times m$. This case represents the worst case scenario for both the number of map tasks to be instantiated and the amount of data to be processed. Moreover, since each pair of input partitions can contain data from any region of the space, there could be some map tasks that produce a huge number of intersecting pairs, while others could have to compare only very far geometries that do not partecipate to the result.

A first improvement of DJNI is represented by the DJGI algorithm which starts from indexed data. DJGI stands for Distributed Join with Grid Index, in this case both input datasets are assumed to be previously repartitioned by using one of the available spatial indexes: the most suitable one given the dataset characteristics [3]. In this case, each partition is identified by a global MBR (Minimum Bounding Rectangle) and map tasks are instantiated only for those pairs of partitions that have a not empty MBR intersection, potentially reducing the number of map tasks and the number of comparisons to be performed inside them.

Fig. 1 illustrates the application of DJNI and DJGI to the same pair of datasets containing rectangles: dataset D_i contains the blue rectangles, while D_j contains the orange ones, for both datasets the corresponding MBR is also depicted. As you can notice for both algorithms, the input datasets have been subdivided into a certain number of partitions (or splits), but while for DJNI each split can contain data coming from any zone of the space, for DJGI each split contains only data that are located nearby, reducing both the number of split combinations to be considered and the amount of unnecessary comparisons to be done. This is a consequence of the fact that the splits of DJNI cover almost the whole MBR of the dataset (see $split_i$ and $split_j$ rectangles in Fig. 1), while the splits (cells) of DJGI cover a small part of the reference space (see c_i and k_j cells in Fig. 1), thus reducing the number of geometric intersections to compute.

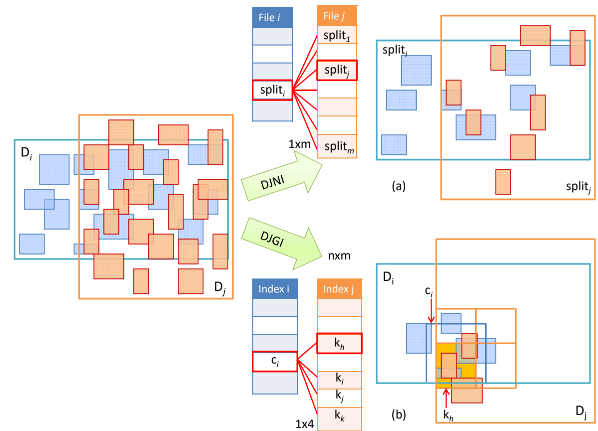

Figure 1: Example of execution of DJNI and DJGI on the same pair of datasets.

As you can notice, when the DJGI algorithm is applied, the dimension of each combined split can be very different from each other, depending on the size and shape of the intersection between the original splits. In order to promote the definition of balanced map tasks, the DJRE algorithm has been developed. DJRE stands for Distributed Join with Repartition, in this case only one of the two input datasets has been previously partitioned by using the most appropriate spatial index, while the other one is repartitioned by using the subdivision (or grid) induced by the first one. In this case, the number of map tasks to be instantiated is equal to the number of partitions of the previous dataset that intersect also the other one. Moreover, while the shape of the obtained splits is uniform, there could be great differences in the number of objects contained in each split, particularly if the two datasets covers only a partial overlapping space, or they cover the same space with different distributions.

SJMR is the only spatial join algorithm which does not assume that the data have been preliminary partitioned with respect to spatial criteria, but it performs itself the best subdivision by considering both datasets together. SJMR stands for Spatial Join MapReduce and it is the MapReduce implementation of the Partition Based Spatial Merge Join [13]. It uses a common global grid for partitioning data before executing the required comparisons. As illustrated in Fig. 2, this grid is regular, namely the shape and dimension of its cells is uniform and is automatically determined

based only on the input file size. The grid definition does not take care of neither the dataset distribution, nor the reference space individually covered by the two input datasets. In this paper, we extend the global partitioning performed by the SJMR at the beginning in order to take care of also these aspects.

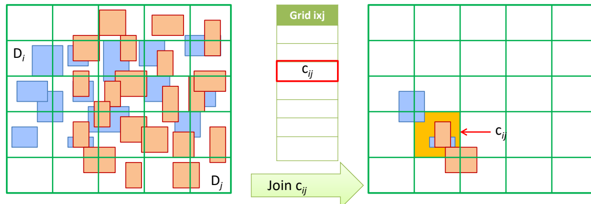

Figure 2: Example of execution of the SJMR algorithm. In this case a global partitioning grid $\mathcal{I}_{\cup}$ is build which includes the union of the MBRs of the two datasets. Each cell c_{ij} is separately processed considering the geometries coming from both datasets.

Tab. 1 provides a brief comparison between the available spatial join algorithms. The first column **Op** reports the name of the spatial join algorithm. Column **BR** indicates if the algorithm requires the use of a modified binary reader in order to load two dataset at time. In particular, a tick indicates that the algorithm considers each dataset individually and contemporarily loads a partition from each of them. Conversely, in the SJMR case, the algorithm uses the default Hadoop reader and is able to load data coming from the two inputs by simply merging the original files. Notice that the modified binary reader induces some additional problems w.r.t. the locality principle: the system guarantees that at least one of the two inputs is locally loaded by the computation node, but the second one can be read both locally or remotely, inducing some overhead. Conversely, with the SJMR each map task is always able to locally read its input, with great advantage in terms of performance of the I/O operations. Column **In** reports the number of datasets that are assumed to be indexed before the spatial join execution, while column **Rep** indicates if a repartition of one dataset is applied before the join. Finally, column **Ref** reports a reference to the original algorithm.

Table 1: Summary of the various spatial join operators.

Op	BR	In	Rep	Ref
DJNI	✓	0	✗	Block Nested Loop Join
DJGI	✓	2	✗	Grid File Spatial Join algorithm [11]
DJRE	✓	1	✓	Bulk-Index Join [14]
SJMR	✗	0	✗	Partition Based Spatial Merge Join [13]

3 MOTIVATING EXAMPLES

In this section we illustrates some example of situations in which the application of the classical partitioning technique provided by the SJMR algorithm can produce unbalanced situations.

We will start by considering the best case regarding the join between two spatial datasets A and B whose geometries are uniformly distributed around the same reference space. This situation is illustrated in Fig. 3.a. The global uniform grid is computed starting from the MBR of the union of the two datasets (i.e., $MBR(A \cup B)$) and by uniformly subdividing the space both horizontally and vertically into $\lceil\sqrt{ds/sp}\rceil$ cells, where ds is the size in bytes of the dataset $A \cup B$, while sp is the size in bytes of the default split size.

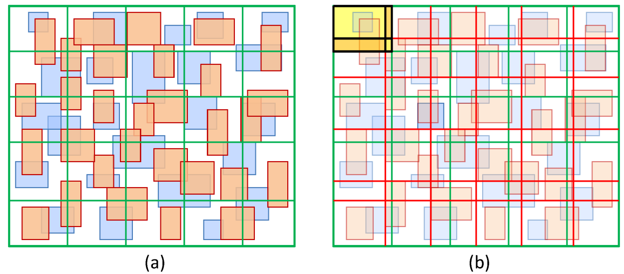

Figure 3: (a) Example of application of SJMR to two completely overlapping datasets. (b) Example of unbalanced grid produced by DJGI: the four left upper cells have been highlighted for showing their different shape and dimension.

This subdivision may be preferable to the application of individual indexes w.r.t. the balancing criteria, because it considers both datasets together. Fig. 3.b illustrates an example in which the splits produced by using separate individual indexes (one in green and one in red) can generate not only a greater number of combinations of intersecting splits, but also very unbalanced combinations, like the ones highlighted in the upper left corner of Fig. 3.b where one green cell is combined with four red cells.

The default construction of the grid in SJMR considers the union of the two datasets. However, if their original reference spaces are not completely overlapping, this grid can contain cells outside the join reference space that include data coming only from one dataset. Some exemplifying situations are illustrated in Fig. 4: in the first case the reference space of one dataset is contained inside the other one, while in the second case the reference spaces of the two datasets are shifted. These situations reveal that it is not necessary to compute the global grid starting from the union of the MBRs of the two datasets, but in order to reduce the amount of unnecessary work, we can instead consider the intersection between the MBRs of the two input datasets during the grid definition. Moreover, in case of uniformly distributed datasets, the definition of the grid starting from the intersection of the MBRs ensures that the obtained splits are also more likely to be balanced.

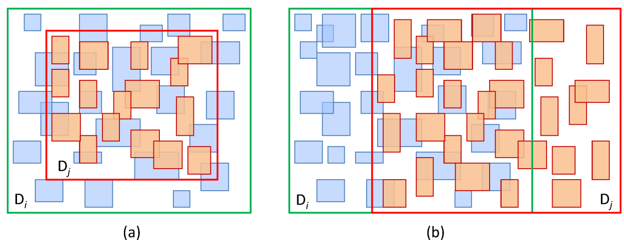

Figure 4: Example of reference spaces that are not completely overlapping: (a) one reference space inside the other, (b) two shifted reference spaces.

Finally, we consider the case in which the two datasets are not uniformly distributed. In this case the use of cells with the same shape and dimension can induce unbalanced split dimensions, as illustrate in Fig. 5.

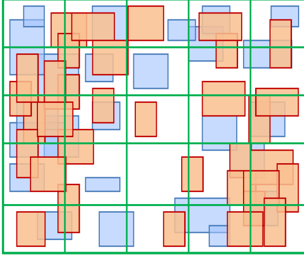

Figure 5: Unbalanced situation: the cells located closer to the boundary are more populated then the central ones.

4 PROBLEM FORMULATION

This section formalizes the problem of obtaining balanced partitions in the context of the spatial join execution.

Definition 4.1 (Spatial Dataset). A *spatial dataset* $D = \{r_1, \ldots, r_n\}$ is a collection of records r_i each one characterized by a spatial attribute g.

In the following, we will use the notation $r_i.g$ in order to denote the spatial attribute g contained in the record r_i.

Definition 4.2 (Partitioning). Given a dataset $D = \{r_1, \ldots, r_n\}$, a partitioning P is a collection of subsets of D:

$$P = \{p_1, \ldots, p_h\} \text{ such that } \forall p_i \in P\,(p_i \subseteq D \wedge D = \cup_i p_i) \quad (1)$$

The general notion of partitioning does not require that any other specific property holds between the contained objects. Conversely, the notion of spatially-enhanced partitioning is defined in order to place nearby objects inside the same partition.

Given a subdivision of the space covered by a dataset D and represented by a set of cells $G = \{c_1, \ldots, c_h\}$, each cell c_k will contain only the records of D whose spatial attribute g has a not empty intersection with c_k. Partitions are then defined starting from this grid subdivision, so that each partition corresponds to a cell of G.

Definition 4.3 (Minimum Bounding Rectangle). Given a geometry g defined by a not empty set of 2D coordinates, i.e. $g = \{(x_1, y_1), \ldots (x_n, y_n)\}$, the *Minimum Bounding Rectangle (MBR)* of g is the rectangle representing the maximum extension of g. In other words, the MBR of g is defined by the coordinates $min(\{x_1, \ldots, x_n\})$, $min(\{y_1, \ldots, y_n\})$, $max(\{x_1, \ldots, x_n\})$, $max(\{y_1, \ldots, y_n\})$. The definition of MBR can be easily extended to a set of geometries as well.

In the following we will use $MBR(g)$, $MBR(D)$ or $MBR(p_i)$ to denote the MBR of a generic geometry g, of a dataset D or a partition p_i, respectively.

Definition 4.4 (Spatially-enhanced partitioning). Given a dataset D and a grid $G = \{c_1, \ldots, c_h\}$ covering $MBR(D)$, a *spatially-enhanced partitioning* P of D is a collection of subsets of D such that:

$$P = \{p_1, \ldots, p_h\} \text{ such that } \forall p_i \in P$$
$$(p_i \subseteq D \wedge D = \cup_i p_i \ \wedge \forall r_j \in p_i (r_j.g \cap MBR(p_i) \neq \emptyset))$$

Independently from the kind of considered partitions (i.e., spatially-enhanced or not), we can define the concept of balanced partitioning as follows.

Definition 4.5 (Balanced Partitioning). A partitioning P for a dataset D is said to be *balanced* if and only if:

$$\forall p_i, p_j \in P : abs(|p_i| - |p_j|) < \varepsilon \quad (2)$$

where $|p_i|$ denotes the cardinality of partition p_i.

The aim of this paper is to obtain balanced spatially-enhanced partitions which contains data from both input datasets involved in the join operation.

5 PROPOSED SOLUTION

This section presents the proposed solution, called EsjMR (Enhanced SjMR), together with a detailed description of the differences introduced w.r.t. the original SjMR algorithm provided by SpatialHadoop.

SjMR is composed of three MapReduce jobs: the first two are responsible for computing the MBR of the two involved datasets separately. The union of these two MBRs is then used by the third job for computing the global uniform grid and performing the spatial join. In particular, as regards to the third job, during the map phase, each mapper assigns its input geometries to one or more cells of the uniform grid, then the reducers (potentially one for each grid cell) receives the geometries contained in a cell and computes the join on them by executing a plane-sweep algorithm. Some expedients are used to avoid the production of duplicated tuples in the final output.

In EsjMR we propose a unique job for computing the MBR of both datasets together, as illustrated in Algorithm 1. In particular, the produced output is already the intersection of the dataset MBRs. Notice that, in order to read the two input dataset, the job uses a strategy already known in literature [6, 15] that combines the two inputs into a unique file by keeping a reference to the source. In the algorithms this additional reference is denoted by representing the value of a tuple as $\langle r_i, f \rangle$ where r_i is the original record contained in the input file, while f with $f = \{1, 2\}$ denotes the fact that the record r_i originally belongs to the first or second input file. Moreover, together with the MBR of the intersection, we produce also an estimation of the number of geometries contained in such intersection. This estimation is useful in order to properly instantiate the partitioning grid, indeed the intersection should contain less geometries than the union of the two original datasets, particularly in the case their reference spaces are not completely overlapping.

In Algorithm 1 each mapper is responsible for updating the dataset MBRs based on the geometries contained in its split, in the psedo-code $r_i.g$ stands for the geometric attribute g contained in the record r_i. Notice that instead of building only two separate MBRs, each mapper maintains and updates two ordered lists of partial MBRs (one for each input file). When a new geometry is processed, the first overlapping MBR intersecting it is updated accordingly, or a new partial MBR is added to the corresponding list. A counter is also maintained for each partial MBR which represents the number of geometries intersecting it. This counter will be used for estimating the number of geometries belonging to the dataset intersection. Clearly, this may be an overestimation, but it is more indicative than considering the sum of the cardinality of the two datasets. At the end of each map, the Cleanup procedure will perform some aggregations of adjacent overlapping MBRs, so that the unique reducer will receive a limited amount of MBRs. The reducer can easily compute both the intersection of the dataset MBRs and an estimation of the number of geometries in the intersection. Notice that in all

Algorithm 1: MBR computation

```
1  class Mapper
2      mbr_i^1 ⟵ mbr_i^2 ⟵ ∅
3      method Map(_, ⟨r_i, f⟩)
4          if f = 1 then
5              if ∃x ∈ mbr_i^1(x.mbr ∩ Mbr(r_i.g)) then
6                  x.mbr ⟵ Extend(x.mbr, Mbr(r_i.g))
7                  x.count ⟵ x.count + 1
8              else
9                  x.mbr ⟵ Mbr(r_i, g)
10                 x.count ⟵ 1
11                 mbr_i^1.SortedAdd(x)
12         else
13             if ∃y ∈ mbr_i^2(y.mbr ∩ Mbr(r_i.g)) then
14                 y.mbr ⟵ Extend(y.mbr, Mbr(r_i.g))
15                 y.count ⟵ y.count + 1
16             else
17                 y.mbr ⟵ Mbr(r_i, g)
18                 y.count ⟵ 1
19                 mbr_i^2.SortedAdd(y)
20     method Cleanup()
21         Compact(mbr_i^1)
22         for x ∈ mbr_i^1 do
23             Write(_, ⟨x.mbr, x.count, 1⟩)
24         Compact(mbr_i^2)
25         for y ∈ mbr_i^2 do
26             Write(_, ⟨y.mbr, y.count, 2⟩)
27 class Reducer
28     mbr_1 ⟵ mbr_2 ⟵ mbr ⟵ EmptyMbr
29     l_1 ⟵ l_2 ⟵ ∅
30     method Reduce(_, ⟨mbr_i, count_i, f⟩)
31         if f = 1 then
32             mbr_1 ⟵ ⟨Extend(mbr_1, mbr_i)
33             l_1 ⟵ l_1 ∪ {(mbr_i, count_i)}
34         else
35             mbr_2 ⟵ ⟨Extend(mbr_2, mbr_i)
36             l_2 ⟵ l_2 ∪ {(mbr_i, count_i)}
37     method Cleanup()
38         mbr ⟵ mbr_1 ∩ mbr_2
39         for x ∈ l_1 ∪ l_2 do
40             if x.mbr ∩ mbr then
41                 count ⟵ count + x.count
42         Write(_, ⟨mbr, count⟩)
```

algorithms, the symbol "_" denotes a dummy serial identifier for a MapReduce input for which we do not take care of.

Given the MBR of the dataset intersection which represents the grid extension, the second task is responsible for performing the balanced partitioning of the two input datasets. It is very similar to the map phase of the second Sjmr task, but it also uses a reduce phase for refining the obtained partitions and producing more balanced splits. This second job is described in Algorithm 2, where it is assumed that each map task knows: (1) the initial

Algorithm 2: Partitioning computation

```
1  class Mapper
2      G ⟵ grid computed using the previous job
3      th ⟵ cell occupation threshold
4      method Map(_, ⟨r_i, f⟩)
5          cl ⟵ IntersectingCells(G, r_i.g)
6          for c ∈ cl do
7              Write(c, ⟨r_i, f⟩)
8  class Reducer
9      method Reduce(c, l = {⟨r_i, f⟩ ...}
10         if |l| > th then
11             ll ⟵ ∅
12             while ll = ∅ ∧ BigSplits(ll) do
13                 ll ⟵ Split(c, ll)
14             for l ∈ ll do
15                 WriteInSplit(l)
16         else
17             WriteInSplit(l)
```

uniform grid G which has an extension equal to the MBR of the datasets intersection and cells with uniform size computed using the dimension (number of objects) of the intersection and the given split size, (2) a threshold value th representing the maximum number of objects to be included in each split.

More specifically, given the dataset $D_\cap$ obtained from the intersection of the two input datasets, whose size in bytes is denoted as $size(D_\cap)$, and the desired size in bytes of a split, denoted as $size(split)$, the estimated initial number of cells of the grid G is computed as $\#cells = \lceil size(D_\cap)/size(split) \rceil$. Given such estimation, the grid G will have the dimension $\lceil \sqrt{\#cells} \rceil \times \lceil \sqrt{\#cells} \rceil$, while the cell weight will be equal to $width(MBR(D_\cap))/\lceil \sqrt{\#cells} \rceil$ and the cell height will be equal to $height(MBR(D_\cap))/\lceil \sqrt{\#cells} \rceil$.

Function IntersectingCells(G, g) returns the set of cells of G which have a not empty intersection with g. This set can be efficiently obtained by considering the MBR of g and thanks to the fixed dimension of the cells. In other words, the index of the first intersected cell is obtained by dividing the minimum x and the minumum y of Mbr(g) for the cell width and height, respectively. In this way the map tasks produce an initial uniform subdivision of the geometries which can be enough if geometries are uniformly distributed. Otherwise, some recursive subdivisions of an overcrowded cell can be necessary.

The reducers, potentially one for each not empty cell produced by the mappers, are responsible for checking the degree of occupancy of such cells. In particular, if the degree of occupancy is less than the given threshold, the geometries contained in the cell can be directly written in a split by the function WriteInSplit(). Otherwise, the geometries inside the cell will be partitioned again by recursively subdividing the cell into four splits (like in a quadtree index). Function Split is responsible for doing such recursive subdivision, while function BigSplit returns true if the degree of occupancy of c overcomes the threshold th.

The last job is responsible for performing the spatial join between the geometries that belong to the intersection, namely the geometries of $D_\cap$. This is done during a map phase, where each mapper receives a split, namely the geometries of both datasets that belong to a particular cell. The first operation to be done is

subdividing the geometries into two lists (one for each dataset) and then the plain sweep algorithm is performed to compute the intersecting pairs. Some expedients can be applied in order to avoid the production of duplicated pairs during the writing performed by the mappers, as already done by SpatialHadoop.

Algorithm 3: Spatial Join

```
1  class MAPPER
2      l1 ⟵ l2 ⟵ {}
3      method MAP( _ , ⟨ri, f⟩)
4          if f = 1 then
5              l1 ⟵ l1.SORTEDADD(ri)
6          else
7              l2 ⟵ l2.SORTEDADD(ri)
8      method CLEANUP()
9          pl ⟵ PLANESWEEP(l1, l2)
10         for ⟨r1, r2⟩ ∈ pl do
11             WRITE(r1, r2)
```

6 EXPERIMENTS AND VALIDATION

This section presents some preliminary results obtained by applying the proposed ESJMR technique, together with a comparison with the partitioning used by SJMR. In particular, we are interested in analysing the degree of balancing obtained by the two partitioning techniques.

Such results are illustrate in Table 2 where two real world cases are considered: (1) the join between the water area (WA) and the primary roads of USA (PR), and (2) the join between the roads (RD) and the administrative subdivisions (AS) of Australia. In the table column $|\text{MBR}_\cup|$ reports the number of geometries contained in the union of the two MBRs, while $|\text{MBR}_\cap|$ the number of geometries in the intersection between the two MBRs. As you can notice, in both cases this number is quite different: the use of the intersection allow to prune unnecessary geometries as soon as possible. Column #splits reports the number of *not empty* cells produced by the two techniques, while column %RDS reports the relative standard deviation between the size of the splits. This last measure is intended as a measure of the balancing degree between the split sizes.

As regards to the number of splits, while SJMR is able to produce less splits (smaller number of join mappers), ESJMR guarantees more balanced parallel tasks. Consider for example the first case: given the original uniform grid, six splits contains a number of geometries less than the desired threshold, while the remaining two contain more than half of the geometries, so they are recursively subdivided by ESJMR.

Table 2: Comparison of the experimental results obtained by applying SJMR and ESJMR algorihtms.

Datasets	SJMR			ESJMR		
	$\lvert\text{MBR}_\cup\rvert$	#splits	%RDS	$\lvert\text{MBR}_\cap\rvert$	#splits	%RDS
WA ⋈ PR	2,305,162	8	181%	2,007,414	22	50%
AS ⋈ PR	1,245,200	5	188%	1,244,800	14	65%

The preliminary experiments encourage the investigation in this direction, since the proposed technique ensures the construction or more balanced splits.

7 CONCLUSION

In recent years the amount of spatial data to be processed is continuously increasing, thanks also to the spread of IoT and mobile devices with geo-spatial capabilities. For this reason, some MapReduce frameworks have been developed which are specifically tailored for modeling and processing spatial data.

Among the possible spatial operation, spatial join is certainly considered a fundamental one for performing meaningful geospatial analysis. However, the join is traditionally considered a critical operation to be performed in the MapReduce context, since it requires to process two distinct datasets (files) at time. For this reason, several join solutions have been developed both in the general case and in the spatial one. As already discussed in literature, the performances of such algorithms greatly depends on the ability to prune unnecessary data as soon as possibile, and to produce balanced partitions. Having balanced partitions means that the parallel map tasks have essentially the same amount of work to do. In this way, we can maximize the benefits induced by the parallelism.

This paper takes inspiration from the SJMR algorithm provided by SpatialHadoop which is the only one that does not require a preliminary construction of a spatial index on both input datasets and also considers both datasets together during the repartition. Starting from this algorithm, we propose an enhanced version of SJMR, called ESJMR, with the aim to produce both more balanced and filtered partitions. Some preliminar experiments have been performed in order to check the benefits of the partitioning induced by ESJMR w.r.t. the one produce by SJMR. Such initial results encourages further development in this direction and will guide the definition of future works.

REFERENCES

[1] A. Belussi, D. Carra, S. Migliorini, M. Negri, and G. Pelagatti. 2018. What Makes Spatial Data Big? A Discussion on How to Partition Spatial Data. In *10th International Conference on Geographic Information Science*. 1–15. https://doi.org/10.1109/ICDE.2015.7113382

[2] A. Belussi, S. Migliorini, and A. Eldawy. 2018. *A Cost Model for Spatial Join Operations in SpatialHadoop*. Technical Report RR108/2018. Dept. of Computer Science, University of Verona. https://iris.univr.it/handle/11562/981957

[3] A. Belussi, S. Migliorini, and A. Eldawy. 2018. Detecting Skewness of Big Spatial Data in SpatialHadoop. In *Proc. of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 432–435. https://doi.org/10.1145/3274895.3274923

[4] A. Belussi, S. Migliorini, M. Negri, and G. Pelagatti. 2015. Validation of Spatial Integrity Constraints in City Models. In *Proc. of the 4th ACM SIGSPATIAL Int. Workshop on Mobile Geographic Information Systems (MobiGIS âĂŹ15)*. 70–79. https://doi.org/10.1145/2834126.2834137

[5] A. Belussi, M. Negri, and G. Pelagatti. 2006. Modelling Spatial Whole-Part Relationships using an ISO-TC211 Conformant Approach. *Inf. Softw. Technol.* 48, 11 (2006), 1095–1103. https://doi.org/10.1016/j.infsof.2006.02.002

[6] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. 2010. A Comparison of Join Algorithms for Log Processing in MapReduce. In *Proc. of the 2010 ACM SIGMOD Int. Conf. on Management of Data*. 975–986. https://doi.org/10.1145/1807167.1807273

[7] A. Eldawy, L. Alarabi, and M. F. Mokbel. 2015. Spatial Partitioning Techniques in SpatialHadoop. *Proc. VLDB Endow.* 8, 12 (2015), 1602–1605. https://doi.org/10.14778/2824032.2824057

[8] A. Eldawy and M. F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *2015 IEEE 31st International Conference on Data Engineering*. 1352–1363. https://doi.org/10.1109/ICDE.2015.7113382

[9] A. Eldawy and M. F. Mokbel. 2017. Spatial Join with Hadoop. In *Encyclopedia of GIS*. Springer, 2032–2036. https://doi.org/10.1007/978-3-319-17885-1_1570

[10] J. Gu, S. Peng, X. S. Wang, W. Rao, M. Yang, and Y. Cao. 2014. Cost-Based Join Algorithm Selection in Hadoop. In *15th Int. Conf. on Web Information Systems Engineering*. 246–261. https://doi.org/10.1007/978-3-319-11746-1_18

[11] L. Harada, M. Nakano, M. Kitsuregawa, and M. Takagi. 1990. Query Processing for Multi-Attribute Clustered Records. In *Proceedings of 16th Int. Conf. on Very Large Data Bases*. 59–70.

[12] S. Migliorini, A. Belussi, E. Quintarelli, and D. Carra. 2020. A Context-based Approach for Partitioning Big Data. In *Proceedings 23rd International Conference on Extending Database Technology (EDBT)*. 1–4. https://doi.org/10.1109/ICDE.1999.754937

[13] J. M. Patel and D. J. DeWitt. 1996. Partition Based Spatial-merge Join. *SIGMOD Rec.* 25, 2 (June 1996), 259–270. https://doi.org/10.1145/235968.233338
[14] J. van den Bercken, B. Seeger, and P. Widmayer. 1999. The bulk index join: a generic approach to processing non-equijoins. In *Proceedings 15th Int. Conf. on Data Engineering*. 257–. https://doi.org/10.1109/ICDE.1999.754937
[15] T. White. 2015. *Hadoop: The Definitive Guide* (4th ed.). O'Reilly Media, Inc.
[16] J. Yu, J. Wu, and M. Sarwat. 2015. 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*. 70:1–70:4. https://doi.org/10.1145/2820783.2820860
[17] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, and et al. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664