Noname manuscript No.
(will be inserted by the editor)

Cost Estimation of Spatial Join in SpatialHadoop

A. Belussi · S. Migliorini · A. Eldawy

Received: date / Accepted: date

Abstract Spatial join is an important operation in geo-spatial applications, since it is frequently used for performing data analysis involving geographical information. Many efforts have been done in the past decades in order to provide efficient algorithms for spatial join and this becomes particularly important as the amount of spatial data to be processed increases. In recent years, the MapReduce approach has become a de-facto standard for processing large amount of data (big-data) and some attempts have been made for extending existing frameworks for the processing of spatial data. In this context, several different MapReduce implementations of spatial join have been defined which mainly differ in the use of a spatial index and in the way this index is built and used. 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. The aim of this work is to deeply analyse them and define a cost model for ranking them based on the characteristics of the dataset at hand (i.e., selectivity or spatial properties). This cost model has been extensively tested w.r.t. a set of synthetic datasets in order to prove its effectiveness.

Keywords Spatial join · cost model · SpatialHadoop · MapReduce · Big spatial data analysis

1 Introduction

In the last few years a large amount of effort has been devoted by researchers to provide a MapReduce implementation of several operations that are usually required for performing big data analysis. In particular, the join operation has

A. Belussi · S. Migliorini
Dept. of Computer Science, University of Verona, Italy E-mail: {name.surname}@univr.it

A. Eldawy
Dept. of Computer Science, University of California Riverside, USA E-mail: eldawy@ucr.edu

attracted much attention, since it is frequently used in data processing, for instance a join is necessary for linking log data to user records. This effort has produced a set of different MapReduce implementations of the join operation [7,15], each one applicable to a particular situation. Therefore, several works have followed in order to propose some sort of heuristics, which allow the system to decide which implementation to apply, given some parameters that characterize the specific case. More specifically, starting from a set of parameters describing both the operation to perform (target parameters) and the input datasets (data parameters), such heuristics are able to produce an estimation of the cost for executing it on a cluster with a given configuration (system parameters). This estimation engine is usually called *cost model*. Only few studies are available in literature which propose a cost model for MapReduce implementations of the join operation, like [7,15].

This paper concentrates on a particular kind of join, called *spatial join*, namely a multi-dimensional join specifically tailored for spatial data [13]. Spatial join is particularly important in GeoScience applications, since it allows to combine several layers with geospatial information, e.g., city boundaries and school district boundaries, and it is also the basis of the map overlay functionality in GIS software. Due to its complexity, many spatial join algorithms have been developed for big data platforms. The performance of these algorithms widely differ according to the characteristics of the input datasets, e.g., their input size and whether or not they are indexed, and also of the cluster, e.g., number of available machines.

Motivating Example – In order to introduce the reader to the problem we aim to address in this paper, we present the following example regarding a geographer (Mary) that needs to compute the spatial join between two huge datasets with the aim to identify the portions of the main roads of USA that might be subject to a flooding risk. A colleague of her (Bob) is doing a similar task in Australia in order to find out the density of the road network in each state of the country, and again the spatial join between two huge datasets has to be computed. Fig. 1 and 2 show the datasets considered in both cases.

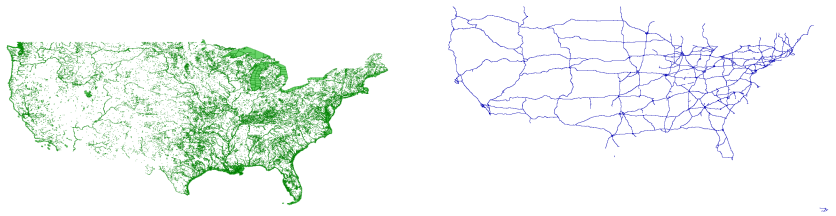

Fig. 1 Case 1: spatial join between the main roads and the water area in USA.

In both contexts the adoption of a big data solution for performing the operation as a MapReduce job can be a good choice, so both Mary and Bob install a Hadoop-based system (the issue about which system to choose accord-

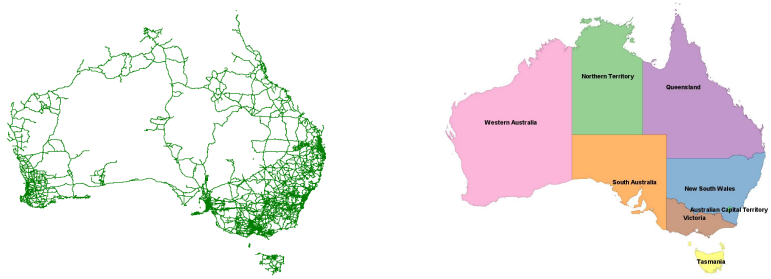

Fig. 2 Case 2: spatial join between the main roads and the states of Australia.

ing to the context will be addressed in the next subsection) and, after loading the datasets on the Hadoop File System (HDFS), they call the method for computing the spatial join, with no clue about which spatial join algorithm to use. Indeed there can be different available implementations of the required join operation and they need to choose one of these. Available algorithms can be for instance: the distributed join with no index (DJNI), the distributed join working on indexed datasets (DJGI), the distributed join with a preliminary repartition phase for rebuilding the index on one dataset (DJRE), and the MapReduce implementation of the partition-based spatial merge join (SJMR). So both Mary and Bob try one randomly. In order to evaluate the impact of this choice, we show in Tab. 1 the performance of the different spatial join implementations when applied to both cases. Experiments were performed in SpatialHadoop, since it allows very easily to select the algorithm to apply. We can observe that: (i) the performance of the algorithms are different, so it is worth looking for the best one; (ii) the best one is not always the same: the best choice is DJRE in Case 1 and DJGI in Case 2; (iii) the performance depends on different factors: on one hand the characteristics of the two involved datasets have an impact, on the other hand also the complexity of the applied algorithms and the implementation choices in MapReduce have to be considered and finally the cluster resources (e.g., the number of nodes, the available memory and disk, and so on) contribute to the final execution time.

The aim of this paper is to identify a set of estimation formulas, namely a *cost model*, that allows us to produce an estimate of the cost for each spatial join algorithm available in a MapReduce system, in order to choose the faster one. Such estimate depends on: (i) some statistics about the input datasets

Spatial join version	Total time			
	Case 1 (Mary)		Case 2 (Bob)	
	seconds	minutes	seconds	minutes
DJNI	473	8	30,908	515
DJGI	426	7	31,296	521
DJRE	1261	21	31,466	524
SJMR	900	15	11,176	186

Table 1 Performance of various spatial join algorithms applied to Cases 1 and 2.

(*input parameters*), (ii) the characteristics of the cluster that runs the experiments (*system parameters*), and (iii) the details of the MapReduce implementation of the algorithms (*implementation details*). The proposed cost model could become the building block of a spatial query optimizer for big data systems, namely the identified formulas can be implemented inside a MapReduce framework and executed before the spatial join operation in order to determine the best algorithm to apply.

System choice – The objective of this paper is to propose a technique for effectively estimating the cost of different MapReduce implementations of the spatial join. Therefore, the chosen system has to provide different implementations of such operation with the additional requirement that the user can autonomously decide which one to apply. Moreover, different kinds of partitioning techniques (indexes) should be available, so that different combinations of indexes and spatial join implementations can be easily tested together.

Currently, two families of systems for the parallel execution of MapReduce jobs can be considered: the Hadoop-based family and the Spark-based family. The former is the older one and provides a pure implementation of the MapReduce programming model, with less expensive requirements in terms of resources, eventually at the expense of performances. Conversely, the latter is the newer one, it includes some optimizations to the classical approach, thus it usually provides a greater efficiency, but with higher requirements in terms of memory in each nodes of the cluster. In general, providing a precise cost model for a parallel spatial join execution has an intrinsic complexity, since it combines both traditional estimation strategies, more tailored on dataset characteristics, with other intertwined factors such as the architecture of the MapReduce execution engine and the cluster configuration. Dealing with complex optimization strategies from the early beginning can be detrimental. Therefore, we choose to concentrate on a more pure MapReduce approach, namely on Hadoop-based solutions, leaving the complexity of Spark optimizations to further investigations. Notice that, the results proposed in this paper can be a good starting point for extending the approach also to the Spark-based family of systems. The choice to concentrate on Hadoop-based solutions has not to be intended as a limitation of the usefulness and importance of this work. Indeed, we remark that the Hadoop-based solutions are particularly effective when the cluster is characterized by nodes having a reduced amount of memory at disposal, or when the datasets are so "big" that cannot fit in the distributed main memory of Spark. At this regard, some works can be found in literature that perform an interesting comparison between Hadoop and Spark concluding that the latter is faster than the former as long as the memory size is big enough for the data size, but as the data size increases becoming bigger than the memory cache, the considered Hadoop cluster outperforms the Spark one [16]. Moreover, as expected, Spark has a higher utilization of

memory resources than Hadoop [28], but in some cases it has also an higher mean disk and network utilization which is instead quite unexpected [22].

MapReduce paradigm – The definition of the proposed cost model greatly follows the architecture of the MapReduce programming paradigm which is illustrated in Fig. 3. The MapReduce paradigm is a processing technique specifically developed for processing huge amount of data in an efficient way. In particular, it requires to subdivide the desired analysis operation into two main subsequent phases: the *map* and the *reduce* phase. During the map phase the various computational nodes perform in parallel and independently from each other the same operation on a particular chunk of the input data, producing a set of intermediate results. These intermediate results are locally written inside each node after each map task concludes, the intermediate results are transferred over the network and sorted by the *shuffler* in order to feed the reducers. This operation can be considered an additional intermediate phase that is transparently performed by the framework in order to connect the two main phases. Finally, during the reduce phase, the intermediate results are combined to produce the final one that is written in the distributed file system. In Fig. 3, the green boxes denote read and write operations performed using a distributed file system, while the white boxes denote local I/O operations. A distributed file system is a file system built starting from the resources of the cluster nodes and shared among them. In particular, Hadoop uses the so called HDFS (Hadoop Distributed File System), in which data are automatically subdivided into independent chunks and distributed among the nodes with a given rate of redundancy (typically 3 copies are produced for each chunk)

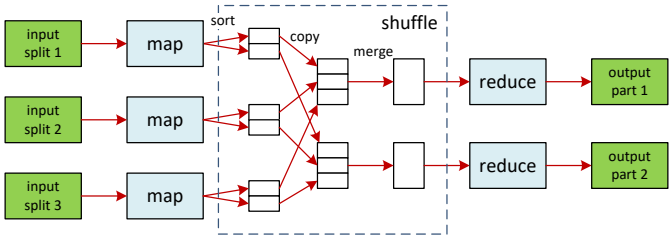

Fig. 3 Schema of a Map Reduce job.

Further details about the MapReduce paradigm and the Hadoop framework can be found in [32]. The extension to more advanced patterns can be intended as a future work that could start from the proposed cost model.

Cost model in summary – The cost model proposed in this paper can be used to deal with the four different variants of spatial join generally available in big data platforms [12]. More specifically, these considered variants are available in many big spatial data platforms, such as SpatialHadoop [11], GeoSpark [34], Simba [33], and Sphinx [14]. They may be classified as: dis-

tributed join with no index (DJNI), distributed join with grid-based index (DJGI), distributed join with repartition (DJRE), and the MapReduce implementation of the partition-based spatial merge join (SJMR).

Several different factors have to be considered during the definition of the mentioned cost model. First of all, some initial considerations have to be formulated about the cost of the spatial join operation by adapting the results, based on the selectivity estimation, coming from previous works, like [19]. Another important factor to consider is the binary nature of the spatial join, namely the impact of transferring the second dataset over network, since the MapReduce framework was originally born for parallel processing *one* big input file at a time. At this regards, the locality of data, i.e., the probability that a job is executed on same node where the data reside, has also to be taken into account. Finally, the impact of the partitioning technique that divides the input data into several splits with or without spatial criteria will be considered. In order to give a general idea about the impact of different partitioning techniques on the implementation of the spatial join in a MapReduce context, we show in Fig. 4 the application of two spatial join algorithms to the same pair of datasets containing rectangles: the distributed join with no index (DJNI) and the distributed join with grid-based index (DJGI). The datasets are presented by means of their reference space (i.e., the Minimum Bounding Rectangle containing all their geometries); dataset D_i, with rectangles in blue, and D_j, with rectangles in orange. In both cases, the files containing the geometries are divided into splits and this subdivision (partitioning) affects the number of map tasks to be executed in parallel. Indeed, in the DJNI case, no spatial criterion has guided the partitionioning, so all possible combinations of split pairs have to be considered. Conversely, in the DJGI case, spatial locality has been considered as grouping criterion during the partitioning; so a reduced number of split combinations has to be considered during the map phase.

The main contribution of the proposed approach is a specialization of the cost models already developed for the classical spatial join operation, for their application in a MapReduce framework, considering also the impact of data partitioning (i.e., spatial indexes). Every algorithm mentioned above will be discussed in details in a specific subsection of Sec. 5 taking as reference the MapReduce implementation the one provided in SpatialHadoop. As already mentioned, the choice of concentrating on SpatialHadoop is due to two main factors: (1) it belongs to the Hadoop-based solutions, namely it exploits the pure MapReduce approach, (2) it is the only system that provides an implementation for all four considered algorithms, while the other mentioned systems implement only some of them. In particular, the use of a repartitioning approach or the construction of a global index like the one built by SJMR are rarely available in other systems.

The structure of the proposed cost model greatly depends on the architecture of the MapReduce paradigm previously described and illustrated in Fig. 3. In particular, for each phase, three different metrics are produced: CPU processing, local disk I/O, and network I/O. In this way, the cost model provides a 3×3 cost matrix that characterizes each MapReduce implementation. In

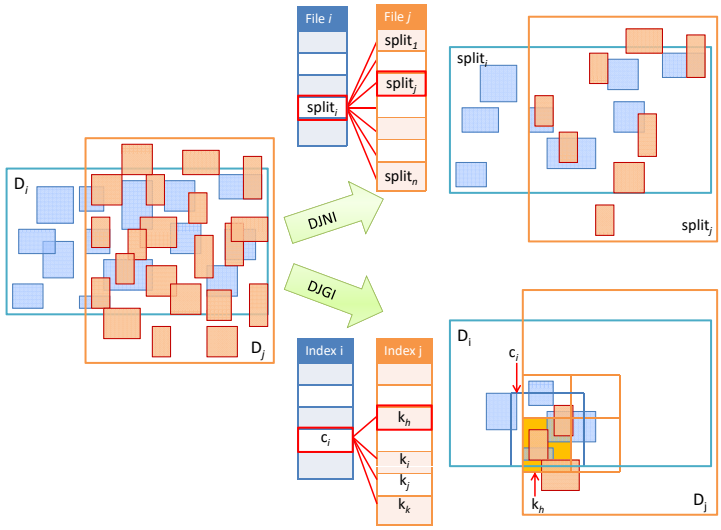

Fig. 4 Example of execution of the spatial join algorithm in the MapReduce framework. The two input datasets are stored respectively into two files (shown on the left). In DJNI each split $split_i$ of the first dataset will be compared with every split of the second dataset. Indeed, a split can contain geometries belonging to any area of the reference space, since no index has been built. Conversely, in DJGI only the splits covering areas with a not empty intersection will be considered.

addition the cost model gives an estimate of the required number of map and reduce tasks (see Sect. 3.2). The main goal of the cost model will be to rank the spatial join implementations according to the matrix of cost estimates, and define a partial order among them introducing a dominance relation, namely a Pareto-set of non-dominated solutions.

The general approach and structure outlined by the proposed cost model, could be applied and extended in order to support other kind of spatial operators, like the k-nn and range query. In particular, the extension to the range query is straightforward and can take advantage of not only the proposed general structure, but also of many of the proposed formulas. Simply, a range query can be modeled as a spatial join where one dataset contains a single geometry that represents the query range. As we will discuss in the following, the definition of a cost model for the spatial join operation contains many critical issues to be solved that are related to the need of processing two datasets at the same time.

We run an extensive experimental evaluation to verify our theoretical cost model. Relatively to the partitioning phase, for each variant of spatial join considered in this paper, we propose two different scenarios: in the first one we suppose that indexes are already available for the input datasets, and in the second one we assume that indexes have to be built. On the data side, we consider different input sets of geometries with increasing cardinalities and containing polygons with different characteristics w.r.t. the size and selectivity

of the geometries (in terms of average MBR area), and the degree of spatial overlapping of the datasets. As previously justified, all the experiments have been executed in SpatialHadoop. The obtained results are reported in Sec. 6 and confirm the effectiveness of the proposed cost model in ranking the alternative algorithms. Indeed, on average the cost model chooses the best algorithm in 89% of the cases.

2 Related Work

To the best of our knowledge, previous work only deals with the cost estimation for implementations in MapReduce of the traditional join operation [7, 15], while relatively to the spatial case only a study about the convenience of partitioning data before joining them has been done [27]. Additional details about related work will be discussed in the following.

Spatial join algorithms – Many algorithms have been defined in literature with the aim to efficiently perform a spatial join between two datasets, considering the cases in which none, one or both inputs have been previously indexed. A comprehensive survey about all these variants can be found in [19]. At the same time, to coupe with all these variants, several benchmarking studies have been performed to evaluate their performance on the basis of the given input [18, 24, 31, 30]. Even if these studies cannot be directly applied to a MapReduce context, they represent a starting point for the definition of the cost model presented in this paper. An extension of the traditional spatial join operation is represented by the multi-way spatial join [21], namely a spatial join where the involved datasets can originate from more than two sources. In this paper, we concentrate on the base case represented by the binary spatial join and left as future work the extension of the formulas to the more general case. This cannot be considered a great limitation, also because available MapReduce systems rarely provides a support for such kind of operation where more than two input files are required.

Cost model for map-reduce join algorithms – The MapReduce framework has become a popular execution environment for the analysis of large amount of textual data. Among all possible processing activities, the join between two inputs is one operation that requires particular attention and has to be carefully studied. In [7] the authors perform an analysis of a number of well-known join strategies in MapReduce and provide an experimental comparison between them. They also explore how the join algorithms can benefit from certain types of preprocessing techniques, such as a repartition phase. Another comparison between join algorithms in Hadoop can be found in [15]. The authors identify the various parts of a join operation and further subdivide them into mappers, shufflers and reducers. An attempt to define an accurate performance model for a generic MapReduce operation has been done in [20]. The authors analyze the composition of MapReduce tasks and relationships

among them, they decompose the major cost items, and presented a vector style cost model which inspired the cost model presented here.

Cost model for map-reduce spatial-join algorithms – A first attempt to define a cost model for spatial-join algorithms in MapReduce can be found in [27] where the authors propose a cost-based and a rule-based optimizer for a generic spatial join. In particular, the authors abstract from any specific implementation and consider a generic spatial join composed of two phases: the partitioning (performed by mappers) and the joining (performed by reducers). The work mostly evaluates the convenience to perform a preliminary partitioning on only one or both datasets. The work in [27] and the one presented in this paper differ essentially for two things: (1) the former starts from experimental results and tries to produce general recommendations using the obtained results; conversely, the latter starts from a precise and detailed cost analysis, producing a set of formulas that are then verified by experiments on synthetic and real-world datasets. (2) The former concentrates only on cluster characteristics during the experimental analysis, while the former is based also on some dataset metadata, such as the selectivity, the dataset cardinalities, and the average number of vertices of the geometries and others.

3 A General Cost Model Framework

This section lays the basis for the definition of a cost model for the various MapReduce spatial join operators mentioned in Sect. 1. In particular, it starts by defining a set of parameters that characterize the execution environment and the input datasets, and then it provides a general notion of cost for a given operator.

3.1 Characterization of the MapReduce Environment

The cost model proposed in this paper has been defined by considering some parameters that characterize the environment in which the spatial join operator is executed. This set of parameters will be called *Hadoop configuration* and is defined as follows.

Definition 1 (Hadoop conf) A Hadoop execution environment is defined by:

- #*nodes*: number of nodes in the cluster.
- #*containers*: number of execution containers in the cluster. The number of containers is typically equal to the number of cores.
- #*parMaps*: maximum number of mappers that can be executed in parallel. This number can be at most equal to the number of containers.

- #*parReds*: maximum number of reducers that can be executed in parallel. We can safely assume that all reducers can be executed in parallel, namely only one reduce step will be performed in the cluster[1].
- *splitSz*: default size of a split provided to a mapper. It usually corresponds to the block size in the Hadoop Distributed File System (HDFS) (128 MBytes by default),
- #*rep*: number of file replications (3 by default).

Moreover, the cost model requires some additional statistical parameters concerning the input datasets which will be called *dataset statistics* and are defined as follows.

Definition 2 (Dataset statistics) Given a dataset D_*, the following parameters regarding the dataset content can be defined (the abbreviation MBR is used to denote the Minimum Bounding Rectangle of a geometry):

- $size(D_*)$: size of the dataset D_* in bytes,
- $\#geo(D_*)$: number of geometries in the dataset D_*,
- $mbr(D_*)$: MBR covering all geometries in D_*,
- $mbrArea^{avg}(D_*)$: given the MBR of all geometries in D_*, it represents the average area of such MBRs,
- $len_x^{avg}(D_*)$ and $len_y^{avg}(D_*)$: given the MBR of all geometries in D_*, they represent the average length on the X and Y axis of such MBRs,
- $\#vert^{avg}(D_*)$: average number of vertices of the geometries in D_*.

Parameters $size(D_*)$ and $\#geo(D_*)$ can be obtained by querying the HDFS. An estimate for parameters $mbr(D_*)$, $len_x^{avg}(D_*)$ and $len_y^{avg}(D_*)$ can be obtained by sampling the input datasets, or we can suppose that they were computed during the scan of the geometries in a previous access, and that the system collects these statistics for refining the quality of the cost model predictions.

For the datasets that have an indexed structure, the following additional parameters are assumed to be known. Notice that this paper concentrates for simplicity on uniform grid-based indexes [23, 26], however the extension to other kind of indexes is straightforward. In particular, as we will discuss in Sect. 4, independently from its kind, any global index is characterized by the construction of grid through which geometries are redistributed among nodes. The main difference resides on the way the grid cells are built and their final shape. In case of a uniform grid index, all cells have the same shape and size.

- $\#cells(\mathcal{I}_*)$: number of cells in a index $\mathcal{I}_*$. Sometimes the abbreviation $\#cells(D_*)$ is used to denote the number of index cells for the dataset D_*.
- $len_x^{cel}(\mathcal{I}_*)$ and $len_y^{cel}(\mathcal{I}_*)$: length on X and Y axis of the cells in the grid index $\mathcal{I}_*$. Sometimes the parameter D_* can be used in place of $\mathcal{I}_*$ to denote its index.

[1] From the official Hadoop documentation, the maximum number of parallel reducers could be set equal to the number of available containers multiplied by a factor of 0.95.

For obtaining more accurate estimations, we define the following two data-related metrics regarding the selectivity between two datasets D_i and D_j.

- $\sigma(A)$: selectivity of the spatial join between two datasets D_i and D_j w.r.t. a reference space A. The selectivity of the spatial join is a real number between 0 and 1 representing the probability that given a pair of geometries (g_i, g_j), such that $g_i \in D_i$ and $g_j \in D_j$, this pair belong to the join result.
- $\sigma^{mbr}(A)$: selectivity of the spatial join between the MBR of the geometries in the datasets D_i and D_j. This selectivity is similar to the previous one, but here the MBRs are considered instead of the real geometries.

These metrics are assumed to be known, indeed they can be estimated in some way using different levels of information about the input datasets [1,2] or in some contexts, system administrators can provide an educated guess based on their experience with the data. In particular, without any knowledge about the dataset characteristics, we can only assume that each geometry might intersect any other geometry in the other dataset. Conversely, by knowing some statistics about the two datasets and assuming a uniform distribution for them, we can obtain a more precise estimation by generalizing the formula proposed in [2] as discussed in [4], obtaining:

$$\sigma(A) \simeq \frac{1}{A}\Big(mbrArea^{avg}(D_i) + mbrArea^{avg}(D_j) + (len_x^{avg}(D_i) \cdot len_y^{avg}(D_j) + len_x^{avg}(D_j) \cdot len_y^{avg}(D_i))\Big) \quad (1)$$

In the experiments, the cost model has been applied by considering the selectivity estimation produced by this formula. Notice that the original formula has been defined for rectangles while the proposed cost model is intended for any kind of geometry. Therefore, the use of σ and σ^{mbr} can produce an overestimation of the real selectivity.

Finally, some additional parameters characterize the size in bytes for storing a vertex of a geometry, an MBR and a record of a dataset: (i) *vertSz* denotes the number of bytes that are needed for the representation of a single vertex; (ii) *mbrSz* indicates the number of bytes required to represent a generic MBR and $recSz(D_*)$ denotes the bytes needed to store a record of the dataset D_*. More specifically, $mbrSz = 4 \cdot vertSz$ and $recSz(D*) = \#vert^{avg}(D_*) \cdot vertSz$.

3.2 Cost of a MapReduce Operator

This section provides a general definition for the cost of a MapReduce operator *op*. The cost of each operator is divided in three components: (i) the cost of the mappers; (ii) the cost of the shufflers and (iii) the cost of the reducers. The shuffle phase is the process through which data is sorted by key and transferred from the mappers to the reducers. Clearly, this phase is performed only if there are some reducers in the considered job.

The cost of each phase is further subdivided in three parts: CPU, local I/O and network I/O. The measurement of these components is based on a set of hypothesis:

- The unit of measure for the CPU cost is the time μ required to compare the x (or y) component of two coordinates, namely to compare two double values: $\mu = \text{time}(\leq (d_1, d_2))$. From this, the time required to test the intersection between two MBRs can be defined as $4 \cdot \mu$, since it requires the comparison of 4 doubles.
- The measure of a disk I/O or network I/O operation is given by the number of bytes read or written.

Given such hypothesis, the cost of a MapReduce operator can be defined as follows:

Definition 3 (Cost of an operator) Given a MapReduce operator *op* and a *Hadoop configuration*, the cost of *op* can be defined by the tuple: $\mathcal{C}(op) = \langle \#map_{op}, \#red_{op}, \mathcal{M}_{op} \rangle$, where $\#map_{op}$ ($\#red_{op}$) is an estimate of the number of mappers (reducers) and $\mathcal{M}_{op}$ is a matrix describing the different cost components (*cpu*, *disk*, *net*) for the different phases (`M`: map, `S`: shuffle, `R`: reduce) of a job:

$$\mathcal{M}_{op} = \begin{bmatrix} \mathtt{M}^{op}_{cpu} & \mathtt{S}^{op}_{cpu} & \mathtt{R}^{op}_{cpu} \\ \mathtt{M}^{op}_{disk} & \mathtt{S}^{op}_{disk} & \mathtt{R}^{op}_{disk} \\ \mathtt{M}^{op}_{net} & \mathtt{S}^{op}_{net} & \mathtt{R}^{op}_{net} \end{bmatrix}$$

Each elements of $\mathcal{M}_{op}$ refers to the cost of an single mapper, reducer or shuffler.

The matrix can be used to obtain the estimated total cost or the estimated effective/parallel cost of a job.

Definition 4 (Estimated total and effective cost) Given a MapReduce implementation *op* of an operator, an *Hadoop configuration* and the tuple $\mathcal{C}(op)$ defining its cost. The estimated total cost of *op* can be obtained by multiplying $\mathcal{M}_{op}$ by a vector containing the estimation of the number of mappers and reducers, as in Eq. 2. In a similar way, the estimated effective cost of *op* can be obtained by multiplying $\mathcal{M}$ by a vector containing the estimation of the number of map and reduce runs, as in Eq. 3.

$$cv_{tot} = \mathcal{M}_{op} \times \begin{bmatrix} \#map_{op} \\ \#red_{op} \\ \#red_{op} \end{bmatrix} = \begin{bmatrix} cpu_{tot} \\ disk_{tot} \\ net_{tot} \end{bmatrix} \tag{2}$$

$$cv_{par} = \mathcal{M}_{op} \times \begin{bmatrix} \#mapRuns \\ \#redRuns \\ \#redRuns \end{bmatrix} = \begin{bmatrix} cpu_{par} \\ disk_{par} \\ net_{par} \end{bmatrix} \tag{3}$$

#*mapRuns* has a lower bound equal to $\lceil \#map_{op}/\#parMaps \rceil$, where #*parMaps* denotes the maximum number of mappers that can be executed in parallel using the current Hadoop configuration, while #*redRuns* is bound by $\lceil \#red_{op}/\#parReds \rceil$, which usually produces only one phase for the reducers. Sect. 4-5 present the cost model for each spatial join operator *op* available in SpatialHadoop. The cost model focuses on the estimation of $\mathcal{C}(op)$, from which we can obtain both cv_{par} and cv_{tot}.

As already mentioned in Sect. 1, the proposed cost model produces a partial order between the various spatial join algorithms, namely a Pareto-set of non-dominated solutions. A total order between them can be obtained only by assigning a weight to the various cost components (CPU, local I/O and network I/O), such weight strictly depends on the cluster characteristics and can be experimentally determined in each single configuration. Given a vector of weight and the estimation $\mathcal{C}(op)$, we can obtain the *total time* of each algorithm in the following way.

Definition 5 (Estimated total time) Given a MapReduce implementation *op* of an operator, an *Hadoop configuration*, a vector cv_{par} defining its effective cost and a vector of weights $W = [w_{cpu}, w_{disk}, w_{net}]$ relating the various cost components. The estimated total time of *op* can be computed as follows:

$$T = \|W \times cv_{par}\|_1 \tag{4}$$

where cv_{par} is defined in Equation 3 and $\| \cdots \|_1$ is the $L1$ norm.

In Sect. 6 we provides a comparison between the various algorithms by using both the partial order induced by $\mathcal{C}(op)$ and the total order obtained with the computation of the total time.

3.3 Processing two Inputs in MapReduce

The join is an operation that requires particular attention when performed in MapReduce, since it needs to process two datasets (files) at time, while Hadoop traditionally processes only one argument. In [4] we describe the details of the reader used by SpatialHadoop to process two input files at time generating compound splits. The use of this reader induces another issue to solve, namely the fact that it is not guaranteed that both splits, composing a compound split and containing the geometries to be joined, reside in the same node. In order to minimize the network I/O cost, the reader tries to put in the same compound split data residing in the same node; in this way a mapper can be allocated to that node and read the split locally. However, when this is not possible, a mapper is assigned by Hadoop to a node where at least one of the two splits resides. In the cost model we need to estimate the local and the network I/O costs, thus given a node n chosen for the execution of a mapper which contains a replica of a D_i split, it is necessary to introduce the probability $\mathcal{P}_{loc}$ that

also the split of D_j is located in n. This may be computed as:

$$\mathcal{P}_{net} = \frac{\binom{\#nodes-\#rep}{\#rep}}{\binom{\#nodes}{\#rep}}, \qquad \mathcal{P}_{loc} = 1 - \mathcal{P}_{net} \tag{5}$$

where $\mathcal{P}_{net}$ is the ratio between the combinations corresponding to an allocation of the replicas of the second split on nodes that do not contain replicas of the first split and all the possible combinations in which the replicas of the second split can be allocated.

4 Spatial Index in SpatialHadoop

Before proceeding with the analysis of the various spatial join operators, we first analyze the concept of spatial index in such environment and how it is built. This section is useful to completely understand and compare the spatial join operators, since some of them make direct use of indexes, while others work without them.

SpatialHadoop has two level of indexes [10]: a global and a local one. The global index determines how data is partitioned among nodes, while the local index determines how data is stored inside each block. In particular, the construction of a global index on a input dataset D, determines that D is stored as a set of data files each one containing the records belonging to one cell (or partition). Some spatial join operators are able to exploit the use of a global index in order to efficiently retrieve the data to be processed.

SpatialHadoop provides different kinds of global indexes or partitioning techniques which can be classified into three main groups: based on space (grid and Quad-tree), based on data (STR, STR+, K-d tree), or based on space filling curves (Z-curve, Hilbert curve) [10]. The selection of the global index to apply is usually left to the user, only in [5] a first heuristic is proposed for automatically selecting the best partitioning technique on the basis of the dataset distribution. Notice that independently from the kind of considered index, the application of a partitioning technique consists in the definition of a grid composed of a certain number of cells which are used for subdividing the dataset geometries. Therefore, the main difference between the various indexes resides on the way such cells are built, namely on their number, shape and dimension. 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 times, absorbing such initial cost. In case of a uniformly distributed dataset, any partitioning technique will produce the same subdivision of the data, so we can safely concentrate on the more simpler and cheaper to construct, which is the uniform grid index. The extension to other kind of indexes is straightforward, since the general job structure is the same and only few estimation formulas have to be modified.

The construction of an index involves two MapReduce jobs, the first one determines the grid to be used for the dataset partitioning (see Sect. 4.1), while the second one partitions the data using the computed grid (see Sect. 4.2).

4.1 Grid Construction

The index structure studied in this paper is the grid one which partitions the data according to a uniform grid. A record that overlaps multiple grid cells is replicated in all these cells. The grid is defined based on the minimum bounding rectangle (MBR) of the input dataset. To compute the MBR, a MapReduce job called MBR is executed. During the map phase, the algorithm computes the MBR of each geometry inside the splits. Thanks to an intermediate combiner, the reducer receives only one MBR from each mapper and generates the final MBR covering the whole dataset.

The cost for operator MBR can be defined as $\mathcal{C}(\text{MBR}) = \langle \#map_{\text{MBR}}, 1, \mathcal{M}_{\text{MBR}} \rangle$ where $\#map_{\text{MBR}} = \lceil size(D_*)/splitSz \rceil$, since the data contained in D_* are partitioned among mappers according to the split size, while only one reducer is used to obtain the final result. The estimation of $\mathcal{M}_{\text{MBR}}$ is discussed below.

Table 2 Estimation of the components of the matrix $\mathcal{M}$ for the grid index construction. Column MBR regards the operator which computes the dataset MBR, while column PART regards the operator which partitions the dataset across the grid.

	Cost	MBR	PART
Map (M)	$\#map$	$\left\lceil \frac{size(D_*)}{splitSz} \right\rceil$	$\left\lceil \frac{size(D_*)}{splitSz} \right\rceil$
	cpu	$\overbrace{\left\lceil \frac{splitSz}{vertSz} \right\rceil \cdot 2\mu}^{\text{MBR computation}}$	$\overbrace{\#geo_{\texttt{sp}}(D_*) \cdot \#cells(\mathcal{I}_*) \cdot 4\mu}^{\text{intersection check}}$
	$disk$	$\overbrace{splitSz}^{\text{reading}} + \overbrace{mbrSz}^{\text{writing}}$	$\overbrace{splitSz}^{\text{reading}} + \overbrace{(\#pairs_{\texttt{mp}}(D_*, \mathcal{I}_*) \cdot recSz(D_*))}^{\text{writing}}$
	net	0	0
Shuffle (S)	cpu	$\overbrace{\#map_{\text{MBR}} \cdot \log(\#map_{\text{MBR}}) \cdot 2\mu}^{\text{MBR ordering}}$	$\left(\overbrace{\frac{\#pairs(D_*, \mathcal{I}_*)}{\#red_{\text{PART}}}}^{\text{list construction}} + \overbrace{\frac{\#cells(\mathcal{I}_*)}{\#red_{\text{PART}}} \log\left(\frac{\#cells(\mathcal{I}_*)}{\#red_{\text{PART}}}\right)}^{\text{ordering}} \right) \cdot 2\mu$
	$disk$	$\overbrace{\#map_{\text{MBR}} \cdot mbrSz}^{\text{writing}}$	$\overbrace{\frac{\#pairs(D_*, \mathcal{I}_*)}{\#red_{\text{PART}}} \cdot recSz(D_*)}^{\text{writing}}$
	net	$\overbrace{\#map_{\text{MBR}} \cdot mbrSz}^{\text{reading}}$	$\overbrace{\frac{\#pairs(D_*, \mathcal{I}_*)}{\#red_{\text{PART}}} \cdot recSz(D_*)}^{\text{reading}}$
Reduce (R)	$\#red$	1	$\max(1, \min(\#cells(\mathcal{I}_*), \#parReds)$
	cpu	$\overbrace{\#map_{\text{MBR}} \cdot 4\mu}^{\text{MBR enlarge}}$	$\frac{\#cells(\mathcal{I}_*)}{\#red_{\text{PART}}} \simeq 0$
	$disk$	$\overbrace{\#map_{\text{MBR}} \cdot mbrSz}^{\text{reading}} + \overbrace{mbrSz}^{\text{writing}}$	$\overbrace{2 \cdot \left(\frac{\#pairs(D_*, \mathcal{I}_*)}{\#red_{\text{PART}}} \cdot recSz(D_*) \right)}^{\text{reading and writing}}$
	net	$\overbrace{mbrSz \cdot (\#rep - 1)}^{\text{writing}}$	$\overbrace{\frac{\#pairs(D_*, \mathcal{I}_*)}{\#red_{\text{PART}}} \cdot recSz(D_*) \cdot (\#rep - 1)}^{\text{writing}}$

Estimate 1 ($\mathcal{M}_{\text{MBR}}$ estimate)**.** The components of $\mathcal{M}_{\text{MBR}}$ are given in Tab. 2.
cpu rationale. (i) The CPU cost of each mapper corresponds to the computation of the MBR for all geometries in a split, thus it linearly depends on their average number of vertices. The total number of vertices in a split can be estimated by dividing the split size by the dimension of a vertex in bytes ($\lceil splitSz/vertSz \rceil$). (ii) The CPU cost of each shuffler is dominated by the ordering procedure it applies on the MBRs produced by the mappers. Thanks to the use of a combiner, the number of MBRs to be ordered is equal to the number of mappers ($\#map_{\text{MBR}}$). (iii) The reducer only computes the global MBR by scanning the MBRs received from the shuffler.
disk i/o rationale. (i) Each mapper reads locally one split of size *splitSz* and writes locally one MBR of size *mbrSz* for each processed geometry. In particular, $\#geo_{\texttt{sp}}(D_*)$ is an estimates of the number of geometries of D_* contained in a split:

$$\#geo_{\texttt{sp}}(D_*) = \frac{splitSz}{recSz(D_*)} \tag{6}$$

(ii) The shuffler only writes locally one MBR of size *mbrSz* for each mapper. (iii) The reducer reads locally what the shuffler has produced and writes locally one copy of the global MBR of size *mbrSz*.
network i/o rationale. (i) The mappers do not read/write remotely, (ii) the shuffler reads remotely one MBR of size *mbrSz* for each mapper, and finally (iii) the reducer writes remotely ($\#rep - 1$) copies of the result.

Given a global MBR for the entire dataset D_*, the number of grid cells (or partitions) is determined by considering the size of D_* so that the content of each cell can fit inside a split. Since $\mathcal{I}_*$ is an index with replication, namely a geometry can be stored several times if it intersects multiple cells, the dataset size is multiplied by a replication factor α in order to consider such situation.

$$\#cells(\mathcal{I}_*) = \max\left(1, \left\lceil \sqrt{\frac{size(D_*) \cdot \alpha}{splitSz}} \right\rceil^2\right) \tag{7}$$

Notice that the number of required cells is enlarged to obtain a squared grid. This way to define the number and shape of the grid cells is what distinguishes a grid index from other kinds of indexes. Clearly, in order to consider in the cost model other kinds of indexes, for instance a Quad-tree or an R-tree, a more complex subdivision of the D_* MBR is required. However, as already mentioned, in case of datasets characterized by a uniform distribution of their geometries, the final result is the same and such additional complexity is not justified, making the use of a uniform grid the best choice.

4.2 Data Partitioning

The grid built by the previous job is used during the following phase which performs the actual data partitioning. In particular, each mapper receives a

split containing a set of geometries of D_* and all cells of $\mathcal{I}_*$, and it produces as output the pairs $\langle c, g\rangle$ where the geometry g intersects the cell c. In order to evaluate the result produced by the mappers, it is necessary to estimate the average number of cells of $\mathcal{I}_*$ that are intersected by a geometry $g \in D_*$:

$$\#cell^{\cap geo}(D_*, \mathcal{I}_*) = \left\lceil \frac{len_x^{avg}(D_*)}{len_x^{cel}(\mathcal{I}_*)} \right\rceil \cdot \left\lceil \frac{len_y^{avg}(D_*)}{len_y^{cel}(\mathcal{I}_*)} \right\rceil + \beta \tag{8}$$

The formula takes care of both the fact that a geometry can span between multiple cells because its extent on the X or Y axis is greater than the corresponding extent of a cell (first two terms) and/or it crosses a cell boundary (see factor β).

Moreover, to consider the case when some geometries in D_* are completely outside the grid (as we will see in Sect. 5.3), we estimate the number of geometries intersecting the grid by multiplying the number of geometries (i.e., $\#geo(D_*)$) by the factor r_{int}:

$$r_{int}(D_*, \mathcal{I}_*) = \frac{area(mbr(D_*) \cap mbr(\mathcal{I}_*))}{area(mbr(D_*))} \tag{9}$$

It considers the size of geometries negligible w.r.t. the size of the reference space. Moreover, when dataset D_* completely overlaps the grid of $\mathcal{I}_*$, r_{int} is equal to 1. In the typical case, the index $\mathcal{I}_*$ is built starting from the dataset D_* and r_{int} will be equal to 1; anyway, as we will see in Sect. 5.3, there can be some cases in which the grid does not correspond to the MBR of D_*.

The cost of PART can be defined as $\mathcal{C}(\text{PART}) = \langle \#map_{\text{PART}}, \#red_{\text{PART}}, \mathcal{M}_{\text{PART}}\rangle$, where $\#map_{\text{PART}} = \lceil size(D_*)/splitSz \rceil$, since the number of mappers only depends on the input size, while $\#red_{\text{PART}} = \max(1, \min(\#cells(\mathcal{I}_*), \#parReds))$, since the number of reducers can be greater than one only if $\#parReds$ is greater that one with a maximum that is equal to the number of cells in the index $\mathcal{I}_*$. The estimation of $\mathcal{M}_{\text{PART}}$ is discussed below.

Estimate 2 ($\mathcal{M}_{\text{PART}}$ estimate)**.** The components of $\mathcal{M}_{\text{PART}}$ are given in Tab. 2. ***cpu rationale.*** (i) The CPU cost of each mapper is given by the cost of checking the MBR intersection between the geometries in a split and all the cells of its index $\#cells(\mathcal{I}_*)$ (this check costs 4μ), where $\#geo_{\mathsf{sp}}(D_*)$ is estimated using Eq. 6 and $\#cells(\mathcal{I}_*)$ using Eq. 7. (ii) The shuffler combines the results produced by the mappers obtaining a list for each cell and orders such lists based on their key (i.e., the cell geometry). The parameter $\#pairs(D_*, \mathcal{I}_*)$ is an estimate of the number of pairs $\langle c, g\rangle$ produced by all mappers. It can be computed in different ways according to the available statistics, some possible estimates are shown in Tab. 3. In the a priori case a geometry overlaps all grid cells, while using Eq. 8-9 we can obtain a more precise estimate. The number of cells to be ordered by each shuffler is computed by dividing the total number of cells ($\#cells(\mathcal{I}_*)$) by $\#red_{\text{PART}}$. Insertion in the list and the test for ordering cells cost both 2μ. (iii) Finally, the reducer simply writes to the HDFS the result, so that a separate file split is generated for each partition. Therefore,

its CPU cost can be considered negligible.

disk i/o rationale. (i) Each mapper reads locally its split of size *splitSz* and writes locally the resulting pairs $\langle c, g \rangle$ whose number is estimated by parameter $\#pairs_{\texttt{mp}}(D_*, \mathcal{I}_*)$ (see Tab. 3). (ii) Each shuffler writes locally the total number of produced pairs, estimated by $\#pairs(D_*, \mathcal{I}_*)$, divided by the number of reducers ($\#red_{\text{PART}}$) and (iii) finally, the reducers read and write the pairs produced by the shufflers.

network i/o rationale. These estimations regard only the reading phase of the shufflers and the writing phase of the reducers, for which the same considerations done in the previous paragraph apply.

Table 3 Estimates for parameters $\#pairs(D_*, \mathcal{I}_*)$ and $\#pairs_{\texttt{mp}}(D_*, \mathcal{I}_*)$.

Par	Estimate	
$\#pairs(D_*, \mathcal{I}_*)$	***a priori*:**	$\#geo(D_*) \cdot r_{int}(D_*, \mathcal{I}_*) \cdot \#cells(\mathcal{I}_*)$
	***with complete statistics*:**	$\#geo(D_*) \cdot r_{int}(D_*, \mathcal{I}_*) \cdot \#cell^{\cap geo}(D_*, \mathcal{I}_*)$
$\#pairs_{\texttt{mp}}(D_*, \mathcal{I}_*)$	***a priori*:**	$\#geo_{\texttt{sp}}(D_*) \cdot r_{int}(D_*, \mathcal{I}_*) \cdot \#cells(\mathcal{I}_*)$
	***with complete statistics*:**	$\#geo_{\texttt{sp}}(D_*) \cdot r_{int}(D_*, \mathcal{I}_*) \cdot \#cell^{\cap geo}(D_*, \mathcal{I}_*)$

5 Spatial Join Algorithms

SpatialHadoop provides four different operators for performing the spatial join. The main differences between them are: (i) the use of indexed or not-indexed data, (ii) the possibility to repartition one of the two datasets using the index of the other, (iii) the execution of the intersection tests on the map or on the reduce side. All operators share a plane-sweep like algorithm (PS_{algo}) for checking the intersections between two list of geometries. The difference mainly resides in the way they build the two lists.

PS_{algo} firstly orders the geometries in the two lists based on the minimum X coordinate of their MBR. Then given the two ordered lists, it scans them switching from one list to the other one according to the MBR distribution along the X axis. Finally, for each pair of intersecting MBRs, the actual intersection between the underlying geometries is checked. The estimate of the PS_{algo} cost is presented below based on the study in [2].

Estimate 3 (CPU cost of PS_{algo})**.** Given two datasets D_i, D_j and two subsets of their geometries $l_i \subseteq D_i$ and $l_j \subseteq D_j$, with cardinality n_i and n_j, respectively, the CPU cost for executing PS_{algo} on them is estimated as:

$$ps(l_i, l_j, A) = \overbrace{n_i \log(n_i) \cdot \mu}^{\text{ordering } l_i} + \overbrace{n_j \log(n_j) \cdot \mu}^{\text{ordering } l_j} + \overbrace{n_i \cdot n_j \cdot \sigma^{mbr}(A) \cdot 4\mu}^{\text{MBR intersection}} + \overbrace{n_i \cdot n_j \cdot \sigma^{mbr}(A) \cdot T^{\cap}_{geo}}^{\text{geometry intersection}}$$

where A is the area of the reference space used to compute the selectivity in Eq. 1.
Rationale: (i) The cost of the ordering phases depends on the cardinality of the lists and is classically estimated as $n \log(n)$. (ii) The number of MBR comparisons can be estimated by means of the MBR selectivity between the two datasets (i.e., $\sigma^{mbr}(A)$), for which an estimate has been proposed in Eq. 1. (iii) The number of intersection tests between geometries can be estimated using the same parameter. (iii) $T^{\cap}_{geo}$ is the cost of testing the intersection between two geometries using a plane-sweep algorithm applied to their vertices. Therefore, given $v = \#vert^{avg}(D_i) + \#vert^{avg}(D_j)$, $T^{\cap}_{geo} = v \log(v) \cdot 2\mu$.

The following sections provide a brief description of each algorithm and an analysis of its costs. Tab. 4 summarizes the main differences between them and provides a reference to the corresponding section. In the table, column **Reader** indicates the use by the mappers of a binary reader, which accesses two files at time; column **Index** reports the number of datasets that require an index, column **Join-side** indicates if the join is done by the mappers or the reducers, column **Rep.** indicates if a repartition is applied before the join, and finally the column **Sect.** reports the subsection describing the algorithm.

Table 4 Summary of the various spatial join operators.

Op	Reader	Index	Join-side	Rep.	Sect.
DJNI	✓	0	map	✗	5.1
DJGI	✓	2	map	✗	5.2
DJRE	✓	1	map	✓	5.3
SJMR	✗	0	reduce	✗	5.4

5.1 Distributed Join with No Index

The first considered spatial join operator works on two input datasets that are not indexed, it is the MapReduce implementation of the Block Nested Loop Join (BNLJ) and it will be called DJNI in the following. DJNI is a map-only job, namely it has no reducers, and clearly it is classified as a map-side join.

Given two input files F_i, F_j, the map input is prepared by the reader, which generates one pair of splits for each mapper; overall all the pairs of splits belonging to the Cartesian products $F_i \times F_j$ will be considered. Fig. 5 illustrates the behaviour of a mapper of DJNI when it works on a "combined" split $s = (split_i, split_j) \in F_i \times F_j$. It initially loads the content of such splits into two lists, then it applies PS_{algo} for checking the intersection between the geometries in the two lists.

The cost for operator DJNI(D_i, D_j) can be defined as: $\mathcal{C}(\text{DJNI}) = \langle \#map_{\text{DJNI}}, 0, \mathcal{M}_{\text{DJNI}} \rangle$, where $\#map_{\text{DJNI}} = \lceil size(D_i)/splitSz \rceil \cdot \lceil size(D_j)/splitSz \rceil$, since all the

Table 5 Estimation for the spatial join operators (map-side). Notice that $\mathtt{M}_*^{\text{PART}}$, $\mathtt{S}_*^{\text{PART}}$ and $\mathtt{R}_*^{\text{PART}}$ are obtained from column 3 of Table 2 by properly instantiating the input dataset and grid index.

	Cost	DJNI	DJGI	DJRE (REP)	DJRE (`join`)
Map (M)	$\#map$	$\left\lceil \frac{size(D_i)}{splitSz} \right\rceil \left\lceil \frac{size(D_j)}{splitSz} \right\rceil$	$\#cells(\mathcal{I}_i) \cdot \#cells(\mathcal{I}_j) \cdot \mathcal{P}^{grid}_{\cap cells}$	$\#map_{\text{PART}}(D_i, \mathcal{I}_j)$	$\#cells(\mathcal{I}_i) \cdot \#cells(\mathcal{I}_j) \cdot \mathcal{P}^{rep}_{\cap cells}$
	cpu	plane-sweep algorithm: $ps(\#geo_{\mathtt{sp}}(D_i), \#geo_{\mathtt{sp}}(D_j), A)$	filtering phase: $(\#geo_{cl}(D_i, \mathcal{I}_i) + \#geo_{cl}(D_j, \mathcal{I}_j))4\mu +$ plane-sweep algorithm: $ps(\#geo^{sel}_{cl}(D_i, \mathcal{I}_i), \#geo^{sel}_{cl}(D_j, \mathcal{I}_j), A_{\mathtt{mmbr}})$	$\mathtt{M}^{\text{PART}}_{cpu}(D_i, \mathcal{I}_j)$	$\mathtt{M}^{\text{DJGI}}_{cpu}$ $A_{\mathtt{mmbr}} = A_{\mathtt{cl}}$
	disk	read D_i: $splitSz +$ reading D_j: $splitSz \cdot \mathcal{P}_{loc} +$ writing: $joinSz_{\mathtt{map}}(A)$	reading D_i: $cellSz(D_i) +$ reading D_j: $cellSz(D_j) \cdot \mathcal{P}_{loc} +$ writing: $joinSz_{\mathtt{map}}(A_{\mathtt{mmbr}})$	$\mathtt{M}^{\text{PART}}_{disk}(D_i, \mathcal{I}_j)$	$\mathtt{M}^{\text{DJGI}}_{disk}$ $A_{\mathtt{mmbr}} = A_{\mathtt{cl}}$
	net	reading D_j: $splitSz \cdot \mathcal{P}_{net} +$ writing: $joinSz_{\mathtt{map}}(A) \cdot (\#rep - 1)$	reading D_j: $cellSz(D_j) \cdot \mathcal{P}_{net} +$ writing: $joinSz_{\mathtt{map}}(A_{\mathtt{mmbr}}) \cdot (\#rep - 1)$	$\mathtt{M}^{\text{PART}}_{net}(D_i, \mathcal{I}_j)$	$\mathtt{M}^{\text{DJGI}}_{net}$ $A_{\mathtt{mmbr}} = A_{\mathtt{cl}}$
Shuffle		0	0	$\mathtt{S}^{\text{PART}}(D_i, \mathcal{I}_j)$	0
Reduce		0	0	$\mathtt{R}^{\text{PART}}(D_i, \mathcal{I}_j)$	0

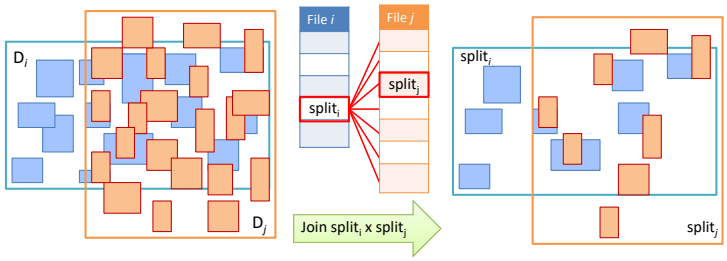

Fig. 5 Example of execution of the DJNI algorithm. The two input datasets D_i and D_j are stored respectively into two files. Each split $split_i$ of D_i will be compared with every split of D_j. A generic split can contain geometries belonging to any area of the reference space, since data is not indexed.

possible pairs of splits are generated and each pair is processed by one mapper. DJNI is a map-only job, thus the number of reducers is 0.

Estimate 4 ($\mathcal{M}_{\text{DJNI}}$ estimate)**.** The components of $\mathcal{M}_{\text{DJNI}}$ are given in Tab. 5.
cpu rationale. Notice that this cost is influenced only by the application of PS_{algo} to the two input lists for which the reference area A is the area of the entire reference space, $A = area(mbr(D_i) \cup mbr(D_j))$. Thus, the cost only depends on the number of geometries contained in the lists $\#geo_{\mathtt{sp}}(D_*)$, which derives directly from the split size (see Eq. 6). The total CPU cost is dominated by the number of mappers ($\#map_{\text{DJNI}}$).
disk i/o rationale. Each mapper certainly reads locally one split of size $splitSz$, and with probability $\mathcal{P}_{loc}$ (see Eq. 5) also the second one. Moreover, it

writes locally the result of the join between the two lists of geometries. The size in bytes of such result (i.e., $joinSz_{\texttt{map}}(A)$) depends on the selectivity between the input datasets and can be estimated as follows where $A = area(mbr(D_i) \cup mbr(D_j))$:

$$joinSz_{\texttt{map}}(A) = \#geo_{\texttt{sp}}(D_i) \cdot \#geo_{\texttt{sp}}(D_j) \cdot \sigma(A) \cdot (recSz(D_i) + recSz(D_j)) \quad (10)$$

network i/o rationale. Each mapper reads the second split of size *splitSz* from the network with a probability $\mathcal{P}_{net}$ (see Eq. 5) and it writes remotely $(\#rep - 1)$ copies of the join result $(joinSz_{\texttt{map}}(A))$.

5.2 Distributed Join with Grid Index

The second spatial join operator considered in this paper works on two indexed datasets, it will be called DJGI and is a MapReduce adaptation of the Grid File Spatial Join algorithm [17]. This operator is similar to the previous one, it is again a map only job (and consequently a map-side join) However, in this case the reader work on indexed data, namely the key of each record represents an index cell, and a filter is used for preparing the input splits, so that only the pairs of splits regarding intersecting cells are generated. Therefore, the number of generated combined splits, and consequently the number of mappers, is equal to the number of pairs of intersecting cells. With reference to Fig. 6, given a cell $c_i \in \mathcal{I}_i$, it is combined only with the cells of $\mathcal{I}_j$ for which the intersection is not empty (i.e., k_h, k_i, k_j, k_k).

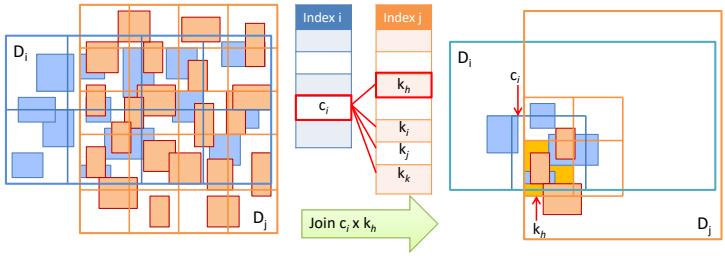

Fig. 6 Example of execution of the DJGI algorithm. The two input datasets D_i and D_j have been indexed using a grid. Each cell c_i of $\mathcal{I}_i$ will be compared only with any other cell of $\mathcal{I}_j$ for which the intersection is not empty (i.e., k_h, k_i, k_j and k_k). In this case only geometries that reside in a nearby space will be compared.

In order to estimate the number of mappers, we need to introduce a formula to compute the probability $\mathcal{P}^{grid}_{\cap cells}$ that given two index cells, $c_i \in \mathcal{I}_i$ and $c_j \in \mathcal{I}_j$, their intersection is not empty. Suppose that the cells of $\mathcal{I}_j$ are smaller

than the cells of $\mathcal{I}_i$, this probability can be defined as:

$$\mathcal{P}^{grid}_{\cap cells} = r_{int}(D_j, \mathcal{I}_i) \cdot r_{int}(D_i, \mathcal{I}_j) \cdot \frac{\left\lceil \frac{len^{cel}_x(D_i)}{len^{cel}_x(D_j)} \right\rceil \cdot \left\lceil \frac{len^{cel}_y(D_i)}{len^{cel}_y(D_j)} \right\rceil \cdot \cancel{\#cells^{\cap}(\mathcal{I}_i, mbr(D_j))}}{\#cells^{\cap}(\mathcal{I}_j, mbr(D_i)) \cdot \cancel{\#cells^{\cap}(\mathcal{I}_i, mbr(D_j))}} \tag{11}$$

where $r_{int}(D_*, \mathcal{I}_\circ)$ has been defined in Eq. 9 and is the percentage of cells of $\mathcal{I}_\circ$ that falls inside the MBR of D_*, while $\#cells^{\cap}(\mathcal{I}_\circ, mbr(D_*))$ is the number of cells of $\mathcal{I}_\circ$ that intersect the MBR of D_* and it can be computed from the available statistics.

The formula is obtained by considering the conjunction of the event corresponding to the choice of a cell of $\mathcal{I}_i$ that falls in the intersection $mbr(D_i) \cap mbr(D_j)$ (namely, $r_{int}(D_j, \mathcal{I}_i)$) with the event corresponding to the choice of a cell of $\mathcal{I}_j$ that falls in the same intersection (namely, $r_{int}(D_i, \mathcal{I}_j)$); then among all the possible pairs of cells that fall in the intersection (denominator), we count the number of intersecting cells (numerator), producing the fraction that appears in the formula.

The cost for the operator DJGI can be defined as $\mathcal{C}(\text{DJGI}) = \langle \#map_{\text{DJGI}}, 0, \mathcal{M}_{\text{DJGI}} \rangle$, where $\#map_{\text{DJGI}} = \#cells(\mathcal{I}_i) \cdot \#cells(\mathcal{I}_j) \cdot \mathcal{P}^{grid}_{\cap cells}$, while the components of $\mathcal{M}_{\text{DJGI}}$ are discussed below. Notice that if the two datasets occupy the same region, namely their MBRs completely overlap, the terms $r_{int}(D_*, \mathcal{I}_\circ)$ are equal to 1 and $\#cells^{\cap}(\mathcal{I}_\circ, mbr(D_*)) = \#cells(\mathcal{I}_\circ)$, so the estimation of the number of mappers becomes: $\#map_{\text{DJGI}} = \lceil len^{cel}_x(D_i) / len^{cel}_x(D_j) \rceil \cdot \lceil len^{cel}_y(D_i) / len^{cel}_y(D_j) \rceil \cdot \#cells(\mathcal{I}_i)$. DJGI is again a map-only job, thus the number of reducers is 0.

Notice that DJGI and all the other algorithms that we will study in the following implement the reference point duplicate avoidance technique [9]. This technique allows to avoid the production of duplicated results even if the same geometry is replicated in several index cells. This is basically a constant-time computation added to each result that prevents the need for a subsequent duplicate check. Therefore, we can safely ignore such problem in the cost model estimation.

Estimate 5 ($\mathcal{M}_{\text{DJGI}}$ estimate)**.** The components of $\mathcal{M}_{\text{DJNI}}$ are given in Tab. 5.
cpu rationale. The CPU cost of a DJGI mapper is very similar to the cost of a DJNI mapper; the only difference is the presence of the preliminary filter phase that reduces the number of geometries that are contained in the lists received by PS_{algo}. This filter is applied at the beginning of each map iteration: the intersection between two cells is computed (called `mmbr`) and only the geometries that intersect `mmbr` are considered during the plane-sweep. The filter cost is dominated by the number of geometries of D_i and D_j contained in each cell of their corresponding indexes $\mathcal{I}_i$ and $\mathcal{I}_j$. The number of geometries in each cell of $\mathcal{I}_\circ$, namely $\#geo_{cl}(D_*, \mathcal{I}_\circ)$, can be estimated, considering a uniform distribution, by dividing the number of geometries in D_* by the number of cells of $\mathcal{I}_\circ$. In the general case, the overlap between a grid $\mathcal{I}_\circ$ and the MBR of D_* can be only partial (as we will see for DJRE in Sect. 5.3), thus in the following

formula we use the number of intersecting cells (i.e., $\#cells^{\cap}(\mathcal{I}_{\circ}, mbr(D_*))$) instead of the total number of cells, even if in the DJGI case they coincide:

$$\#geo_{cl}(D_*, \mathcal{I}_{\circ}) = \frac{\#geo(D_*) \cdot \alpha}{\#cells^{\cap}(\mathcal{I}_{\circ}, mbr(D_*))} \tag{12}$$

The replication factor $\alpha > 1$ introduced in Tab. 3 is also used here to determine the geometries per cell.

The filter phase reduces the number of geometries to be considered by PS_{algo}. In particular, parameter $\#geo^{sel}_{cl}(D_*, \mathcal{I}_{\circ})$ is an estimate of the average number of geometries of D_* that survive after the filter phase. It is obtained by multiplying $\#geo_{cl}(D_*, \mathcal{I}_{\circ})$ by a filter factor, denoted as $\psi(D_*)$, that can be estimated by considering the dimension of the index cells. Let us assume that $\mathcal{I}_j$ has cells that are smaller than the cells of $\mathcal{I}_i$, then $\psi(D_j) = 1$, while $\psi(D_i) = area(\text{cells of } \mathcal{I}_j)/area\ (\text{cells of } \mathcal{I}_i)$. This can be an over-estimation of the selectivity, since it considers only the cell dimensions and not their displacement. For DJGI the estimation of PS_{algo} considers a selectivity computed on an area equal to the average area of the not empty `mmbr` (called $A_{\texttt{mmbr}}$).

disk i/o rationale. As for DJNI, each mapper reads locally the first dataset and with a probability $\mathcal{P}_{loc}$ the second one, the only difference is that the split size is not fixed but it depends on the number of geometries in each index cell, namely $cellSz = \#geo_{cl}(D_*, \mathcal{I}) \cdot recSz(D_*)$. Moreover, it writes locally one copy of the join result (i.e., $joinSz_{\texttt{map}}(A_{\texttt{mmbr}})$) whose dimension depends on the selectivity computed using $A_{\texttt{mmbr}}$.

network i/o rationale. As in the case of DJNI, each mapper reads remotely the second dataset with a probability $\mathcal{P}_{net}$ and the split size is the same of the one used for the local I/O. Moreover, it writes remotely $(\#rep - 1)$ copies of the join result with the same size estimated for the local I/O.

5.3 Distributed Join with Repartition

A variant of DJGI is the operator denoted as DJRE, which additionally performs a repartition of one of the two datasets w.r.t. the index of the other. It is a MapReduce adaptation of the Bulk-Index Join [6]. In particular, if both input datasets are indexed, the smaller one is repartitioned using the index of the bigger one; conversely, if only dataset D_j (D_i) is indexed, then D_i (D_j) is repartitioned using the index of D_j (D_i). This operator consists of two map-reduce jobs, the first one performs the repartition and the second one is similar to DJGI. The repartition phase can be particularly useful when the two datasets have a partial overlapping, in this case the geometries of the repartitioned dataset that do not intersect the MBR of the other one are filtered out, since they certainly will not participate to the join result. Moreover, only pairs of fully overlapping cells will be considered, thus leading to a reduced number of balanced mappers.

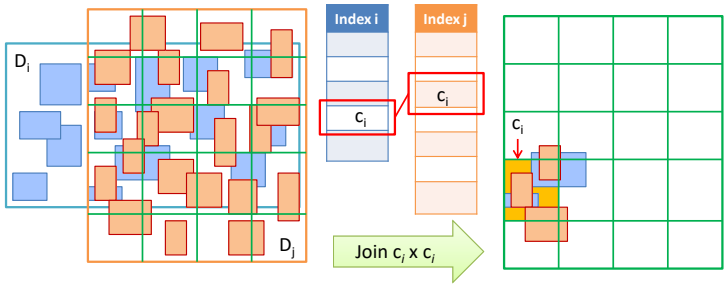

Fig. 7 Example of execution of the DJRE algorithm. In this case dataset D_i is repartitioned using the grid index of D_j. After such repartition any cell $c_i \in \mathcal{I}_i$ will be compared only with the corresponding cell $c_i \in \mathcal{I}_j$.

5.3.1 Repartition Phase

Without loss of generality, we consider that D_i has to be repartitioned w.r.t. the index of D_j. The repartition is performed by a job, called REP, composed of a map and a reduce phase. The mappers scan each geometry of D_i and build a pair $\langle c, g \rangle$ for each cell $c \in \mathcal{I}_j$ that intersects a geometry $g \in D_i$. This job is similar to the PART job performed during the index construction and illustrated in Sect. 4.2. However, in this case the dataset to be partitioned is D_i while the considered index is $\mathcal{I}_j$, namely the index of D_j. Therefore, with reference to Eq. 9, factor r_{int} could be different from 1.

5.3.2 Join Phase

The cost of the join job is the same as the cost of the DJGI operator except for the formula which computes the $\mathcal{P}_{\cap cells}$, since in this case the two datasets share the same index grid. In other words, the number of combined splits that will be generated is equal to the number of cells that intersect the MBR of both datasets. $\mathcal{P}_{\cap cells}$ can be obtained by simplifying Eq. 11 as follows:

$$\mathcal{P}^{rep}_{\cap cells} = \frac{\#cells^{\cap}(\mathcal{I}_j, mbr(D_i))}{\#cells^{\cap}(\mathcal{I}_j, mbr(D_j)) \cdot \#cells^{\cap}(\mathcal{I}_j, mbr(D_i))} \tag{13}$$

Moreover, as regards to the selectivity used for computing the result dimension, namely $joinSz_{\texttt{map}}(A_{\texttt{mmbr}})$, since the cells of both indexes are the same, for each mapper it always occurs that: $A_{\texttt{mmbr}} = A_{\texttt{cl}}$, where $A_{\texttt{cl}}$ is the area of a cell.

5.4 Spatial Join Map Reduce

The last operator considered in this paper is called SJMR (Spatial Join Map Reduce) and has been designed to perform spatial join efficiently for non-indexed datasets. It is the map-reduce implementation of the Partition Based Spatial Merge Join [25] and it will be denoted as SJMR in the following. It uses a

Table 6 Estimation for the spatial join operators (reduce-side).

Cost		SJMR (D_i MBR)	SJMR (D_j MBR)	SJMR (join)
Map (M)	#*map*	$\#map_{\text{MBR}}(D_i)$	$\#map_{\text{MBR}}(D_j)$	$\left\lceil \frac{size(D_i)+size(D_j)}{splitSz} \right\rceil$
	cpu	$\mathtt{M}^{\text{MBR}}_{cpu}(D_i)$	$\mathtt{M}^{\text{MBR}}_{cpu}(D_j)$	$\overbrace{\#pairs_{\text{mp}}(D_\cup,\mathcal{I}_\cup)}^{\text{cell-geom pairs}} \cdot 4\mu$
	disk	$\mathtt{M}^{\text{MBR}}_{disk}(D_i)$	$\mathtt{M}^{\text{MBR}}_{disk}(D_j)$	$\overbrace{splitSz}^{\text{reading}} + \overbrace{\#pairs_{\text{mp}}(D_\cup,\mathcal{I}_\cup) recSz(D_\cup)}^{\text{writing}}$
	net	$\mathtt{M}^{\text{MBR}}_{net}(D_i)$	$\mathtt{M}^{\text{MBR}}_{net}(D_j)$	0
Shuffle (S)	*cpu*	$\mathtt{S}^{\text{MBR}}_{cpu}(D_i)$	$\mathtt{S}^{\text{MBR}}_{cpu}(D_j)$	$2 \cdot (\overbrace{\#pairs_\cup}^{\text{list constr.}} + \overbrace{\#cells^{red}_\cup \log \#cells^{red}_\cup}^{\text{ordering}})\mu$
	disk	$\mathtt{S}^{\text{MBR}}_{disk}(D_i)$	$\mathtt{S}^{\text{MBR}}_{disk}(D_j)$	$\overbrace{\#pairs_\cup \cdot recSz(D_\cup)}^{\text{writing}}$
	net	$\mathtt{S}^{\text{MBR}}_{net}(D_i)$	$\mathtt{S}^{\text{MBR}}_{net}(D_j)$	$\overbrace{\#pairs_\cup \cdot recSz(D_\cup)}^{\text{reading}}$
Reduce (R)	#*red*	$\#red_{\text{MBR}}(D_i)$	$\#red_{\text{MBR}}(D_j)$	$\max(1, \min(\#cells(\mathcal{I}_\cup), \#parReds))$
	cpu	$\mathtt{R}^{\text{MBR}}_{cpu}(D_i)$	$\mathtt{R}^{\text{MBR}}_{cpu}(D_j)$	$\#cells^{red}_\cup \cdot \overbrace{ps(\#geo_{cl}(D_i,\mathcal{I}_\cup), \#geo_{cl}(D_j,\mathcal{I}_\cup), A_{\mathtt{cl}})}^{\text{intersection test}}$
	disk	$\mathtt{R}^{\text{MBR}}_{disk}(D_i)$	$\mathtt{R}^{\text{MBR}}_{disk}(D_j)$	$\overbrace{\#pairs_\cup \cdot recSz(D_\cup)}^{\text{reading}} + \overbrace{\#cells^{red}_\cup \cdot joinSz_{cl}(A_{\mathtt{cl}},\mathcal{I}_\cup)}^{\text{writing}}$
	net	$\mathtt{R}^{\text{MBR}}_{net}(D_i)$	$\mathtt{R}^{\text{MBR}}_{net}(D_j)$	$\overbrace{\#cells^{red}_\cup \cdot joinSz_{cl}(A_{\mathtt{cl}},\mathcal{I}_\cup)}^{\text{writing}} \cdot (\#rep - 1)$

uniform grid for performing the spatial join which is computed from the union of the MBR of the two datasets, while the cell dimension is automatically determined based on the input files size.

5.4.1 Grid Computation

The uniform grid is built by using two map reduce jobs, each one is responsible for determining the MBR of a dataset. The cost of the each MBR job can be estimated as described in Sect. 4.1. The two MBRs are then merged into a global one and the number of required cells is determined so that the content of each cell can fit into a split. In particular, $\#cells(\mathcal{I}_\cup)$ denotes the number of cells for this uniform grid and it is computed using Eq. 7 where $D_* = D_i \cup D_j$.

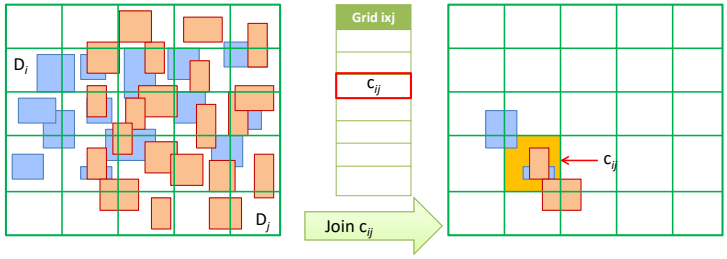

Fig. 8 Example of execution of the SJMR algorithm. In this case a global index $\mathcal{I}_\cup$ is build which includes the union of the MBRs of the two datasets. Each cell c_{ij} is separately processed considering the geometries coming from both datasets.

5.4.2 Join Phase

As regards to the join phase, each mapper receives in input a set of geometries coming from both datasets. Notice that SJMR does not use the binary reader introduced in Sect. 3.3 for combining the input files. Conversely, the input files are merged into a single one by concatenating them. For each geometry g in input, a mapper directly computes the set of cells that intersect its MBR. For each match cell-geometry found in this step, it writes in output the pair $\langle c, \langle f, g \rangle\rangle$, where the key is the grid cell c, and the value is again a pair containing the identifier f of the file from which the geometry comes from and the geometry g itself. Each reducer can work on one or more cells. For each cell, it builds two lists by dividing the geometries in a cell based on the file from which they come from. Given the two lists, PS_{algo} is performed on them for producing the final result. Fig. 8 illustrates the behaviour of SJMR when it processes a grid cell c_{ij} belonging to the global grid built considering the union of the two datasets.

The cost of operator SJMR can be defined as $\mathcal{C}(\text{SJMR}) = \langle \#map_{\text{SJMR}}, \#red_{\text{SJMR}}, \mathcal{M}_{\text{SJMR}} \rangle$, where $\#map_{\text{SJMR}} = \lceil (size(D_i) + size(D_j))/splitSz \rceil$, since it works on the union of the two input datasets, while $\#red_{\text{SJMR}} = \max(1, \min(\#cells(D_\cup), \#parReds))$, since the number of reducers can be greater than one only if the Hadoop configuration allows more than one reducers, with a maximum equal to the number of cells. The components of the $\mathcal{M}_{\text{SJMR}}$ are discussed below.

Estimate 6 ($\mathcal{M}_{\text{SJMR}}$ estimate)**.** The components of $\mathcal{M}_{\text{SJMR}}$ are given in Tab. 6.
cpu rationale. (i) Each mapper works on a split coming from the union of the two input datasets: $D_\cup = D_i \cup D_j$. The operation performed on each geometry takes a constant time (i.e., 4μ) to determine the intersecting cells, since it only uses some comparisons between the MBR coordinates and the cell lengths. The average number of geometries contained in a split is represented by the parameter $\#geo_{\texttt{sp}}(D_\cup)$, which can be estimated as $\#geo_{\texttt{sp}}(D_\cup) = splitSz/recSz(D_\cup)$, where $recSz(D_\cup)$ is the average record size computed considering the records of both datasets. Moreover, the estimated number of matches cell-geometry ($\#pairs_{\texttt{mp}}(D_\cup, \mathcal{I}_\cup)$) is computed as in Tab. 3. (ii) The shufflers collect the pairs produced by the mappers, combines the record related to the same cell into

lists and order such lists based on the key (i.e., the cell). The number of records to be combined by the shuffler is estimated by the parameter $\#pairs_{\cup}$:

$$\#pairs_{\cup} = \frac{\#pairs(D_i, \mathcal{I}_{\cup}) + \#pairs(D_j, \mathcal{I}_{\cup})}{\#red_{\text{SJMR}}}$$

The number of cells ordered by each shuffler is estimated by $\#cells_{\cup}^{red}$:

$$\#cells_{\cup}^{red} = \frac{1}{\#red_{\text{SJMR}}} \cdot (\#cells^{\cap}(\mathcal{I}_{\cup}, mbr(D_i)) + \#cells^{\cap}(\mathcal{I}_{\cup}, mbr(D_j)) - \#cells^{\cap}(\mathcal{I}_{\cup}, mbr(D_i \cap D_j))$$

where the term $\#cells^{\cap}(\mathcal{I}_{\cup}, D_*)$ is the number of cells of $\mathcal{I}_{\cup}$ that intersect D_*. Notice that in the formula $\#cells^{\cap}(\mathcal{I}_{\cup}, mbr(D_i \cap D_j))$ has been subtracted for not counting twice the same cell. (iii) Finally, the reducers perform the spatial join using PS_{algo} inside each cell. Each reducer works on a number of cells equals to $\#cells_{\cup}^{red}$ (see Eq. 14).

disk i/o rationale. (i) Each mapper reads locally a union split of size *splitSz* and writes locally a record for each intersecting pair of geometry-cell. The average number of intersecting pairs has been estimated above by the parameter $\#pairs_{\texttt{mp}}(D_{\cup}, \mathcal{I}_{\cup})$, while the size of each record is estimated as $recSz(D_{\cup}) = \#vert^{avg}(D_{\cup}) \cdot vertSz$, where $\#vert^{avg}(D_{\cup})$ is the average number of vertices of the geometries contained in the union of the two datasets $D_i \cup D_j$. (ii) Given the output produced by the mappers, each shuffler writes locally its combined records whose number is estimated by $\#pairs_{\cup}$ and whose size by $recSz(D_{\cup})$. (iii) Each reducer reads locally the input produced by its corresponding shuffler and writes a portion of the result whose size is obtained by multiplying the number of its cells ($\#cells_{\cup}^{red}$) by $joinSz_{cl}(A_{cl}, \mathcal{I}_{\cup})$

$$joinSz_{cl}(A_{\texttt{cl}}, \mathcal{I}_{\cup}) = \#geo_{cl}(D_i, \mathcal{I}_{\cup}) \cdot \#geo_{cl}(D_j, \mathcal{I}_{\cup}) \cdot \sigma(A_{\texttt{cl}}) \cdot (recSz(D_i) + recSz(D_j)) \quad (14)$$

network i/o rationale. Only the shufflers and the reducers performs network I/O. In particular, each shuffler reads a portion of the data produced by all mappers (of size $\#pairs_{\cup} \cdot recSz(D_{\cup})$) and the reducers remotely write ($\#rep - 1$) copies of the results.

6 Validation and Experiments

The cost model presented in the previous sections has been validated using a set of experiments on synthetic datasets. The choice of using synthetic datasets is justified essentially by two reasons: the need to ensure a uniform distribution of the geometries inside the dataset, and the need to vary in a controlled way the various characteristics discussed in Sect. 3.1. Experiments have been performed on a cluster composed of one master node and three slave nodes in

which 2 containers can be potentially instantiated in parallel, for a maximum number of parallel mappers or reducers equal to 6 (i.e., #*parMaps*, #*parReds*). Notice that the cost model is parametric on the cluster characteristics, different cluster configurations can lead to different rankings in the join algorithms, anyway here we are interested in evaluating the accuracy of the cost model, not the algorithm performances.

We initially evaluate the quality of the cost model by comparing the estimated costs and the actual measured costs, considering a representative case where geometries in each dataset have an MBR area equal to 1e−8 w.r.t. the area of the reference space and five vertices, while the size of a dataset is 128 MBytes and the size of the other one varies from 128 MBytes to 5120 MBytes. During this comparison we try to keep the various cost components as much separated as possible. However, as regards to the I/O, we compare the sum of the local and network I/O estimates with the total number of bytes read and written as reported in the Hadoop logs (`HDFS/File: Number of bytes read/written`); indeed, the statistics that we can find in the logs do not distinguish between network and local I/O performed on HDFS. The actual comparison regarding this overall I/O cost can be done directly, since the estimates and measured values are both in bytes, while for the CPU cost we can only compare the trend of the estimates with the trend produced by the log value `CPU time spent`. Fig. 9 reports the estimated cost for the overall I/O, while Fig. 10 reports the number of bytes read and written by each algorithm as reported in the Hadoop logs. The two trends are very similar: the average difference between the two is about 9%. As you can notice, SJMR has the greater I/O since it initially repartitions both datasets using a common grid, while DJRE has the lower I/O for two reasons: it repartitions only the smaller dataset and then, having a common grid, the number of intersecting cells to read during the join phase is reduced. Similarly, Fig. 11-12 report the estimated and actual CPU costs; again the two trends are very similar. However, the estimated CPU costs are sightly different from the measured one, since the logs include the time required to instantiate the MapReduce jobs, while we omit it in the proposed cost model formulation. Notice that, since the setup cost is uniform for all algorithms, we can safely ignore this additional setup cost, because it will not alter the choice of the best algorithm. Relatively to the CPU, SJMR is the algorithm with the least number of comparisons to perform, while DJNI is the one with the highest CPU cost.

In order to compare the results produced by the cost model with the one obtained from the experiments, we introduce a relation of dominance. This relation produces a partial order among the operators which considers the three cost components in a separate way and can be use used as a first hint to choose the best candidate abstracting from the specific cluster characteristics. In other words, the cost model will produce a Pareto-set of non-dominated solutions.

Definition 6 (Dominance) Given two cost vectors $cv_*(op_1)$, $cv_*(op_2)$ (Def. 4), representing the estimates of the cost for operators op_1 and op_2 respectively,

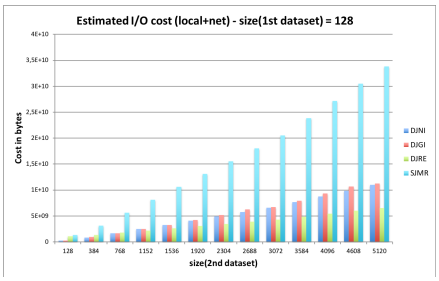

Fig. 9 Estimated trend for the local and network I/O in MBytes.

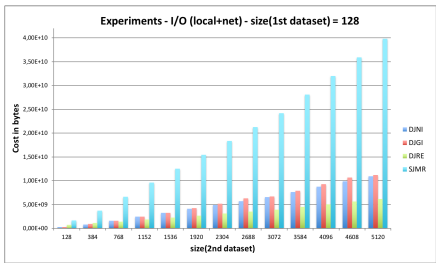

Fig. 10 Experimental trend for the local and network I/O in MBytes.

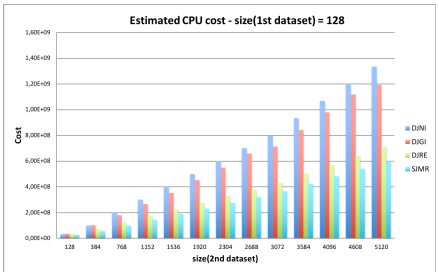

Fig. 11 Estimated trend for the CPU cost.

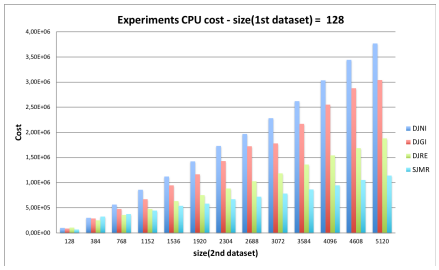

Fig. 12 Experimental trend for the CPU cost.

where * stands for *tot* or *par*, we say that op_1 dominates op_2 according to the cost model ($op_1 \prec op_2$), if the following conditions hold: (i) $\forall i \in \{1,2,3\}$: $cv_*(op_1)[i] \leq cv_*(op_2)[i]$, and (ii) $\exists j \in \{1,2,3\}$: $cv_*(op_1)[j] < cv_*(op_2)[j]$, where $cv_*(op)[i]$ denotes the i-th elements of vector $cv_*(op)$.

The cost model has been tested in various scenarios by varying different characteristics of the involved datasets such as: the cardinality of the two datasets, the dimension of the geometry MBRs and the rate of dataset overlapping. However, as mentioned above, the dominance relation produces only a partial order among the operators and in some cases it is not enough selective to produce a unique choice. For instance, referring to the cases depicted in Fig. 10-12, we noticed that SJMR is the one with the best CPU cost while it is the worst considering the I/O cost, conversely DJRE is the best for the I/O cost and they have higher CPU costs. Therefore, the final choice between them can be performed only considering the specific performance characteristics of the cluster at hand. More specifically, the choice among the algorithms in the Pareto-set can be refined through the formula in Eq. 2 by assigning a different weight to the various cost components. For this reason, we performed a set of micro-benchmarks on the used cluster in order to determine the relation existing between the CPU, local I/O and network I/O costs. Moreover, such micro-benchmarks reveal that the use of the binary reader described in Sect. 3.3 takes more time per byte in producing the next value w.r.t. the Hadoop default reader that processes a single file at time. Therefore, from

these benchmarks we obtain a set of constants that compare the unitary cost (i.e., per byte) of performing a local read using a binary or default reader (br_{io}, sr_{io}), a network read using a binary or default reader (br_{net}, sr_{net}), a local write (w_{io}), or a network write (w_{net}), w.r.t. the cost of a unitary CPU operation (i.e., an MBR comparison). These constants can be used as weights in Eq. 2 in order to combine the three cost components and produce a total order among the members of the Pareto-set.

The remainder of this section describes the various performed experiments and the obtained results using both the dominance relation and the cumulative effective cost. In particular, we compare the effective time taken by the algorithm in each test with the parallel cost introduced in Eq. 3 considering 6 as the maximum number of parallel mapper and/or reducers.

Experiment 1 (Cardinality – ExpCard)**.** Given two datasets with the same reference space (i.e., percentage of overlapping equals to 100%) and whose elements are polygons with the same number of vertices (i.e., 5) and an MBR of size 1e−8 w.r.t. the reference space, in **ExpCard** experiments we changed the cardinality of the two datasets starting from 128 MBytes (1 split) to 5120 MBytes (40 splits).

Tables 7-8 describe the detailed results for one of the groups of cardinality considered during **ExpCard**, in particular the case in which $|D_i| = 3584$ MBytes (i.e., 28 splits). In the tables, column # contains the number of splits for the dataset D_j, whereas the other four columns are relative to each considered join operator and report: (i) the ordering obtained from the experiments as a number, (ii) the set of non-dominated operators returned by the cost model, labelled by a green square, (iii) the operator dominated by all the other ones, labelled by a red square, and (iv) the positions suggested by the cumulative cost between round brackets. For instance, with reference to the first row of Tab. 7, the experiments produce the following order: DJRE, DJGI, SJMR and DJNI, which is represented by the values 4, 2, 1, and 3. The cost model instead produces a Pareto-set containing two algorithms: DJRE and SJMR, the ones with a green cell background. By applying the formula in Eq. 2, the cost model produces the following total order: DJRE, SJMR, DJGI and DJNI. As you can notice, in this case there is not an algorithm dominated by all the others, conversely, in all the other rows there is an algorithm with a red cell background. Tab. 7 does not consider the cost of index construction, while Tab. 8 considers also the cost of index building (+2in or +1in mean that an algorithm requires the preliminary construction of two or one index, respectively). Further results computed considering different cardinalities can be found in [4].

As you can notice, the suggestions obtained considering also the cost of index construction are more accurate, since this additional cost produces a clearer separation between the operators. Conversely, Tab. 7 reveals that the dominance relation is not always sufficiently selective to perform a choice, while the cumulative cost can sometimes suggest an algorithm which is not the best, but is for instance the second one. This is particularly true when DJGI

Table 7 Results of **ExpCard** without consider the index cost for $|D_i| = 28$ splits.

#	DJNI	DJGI	DJRE	SJMR
1	4(4)	2(3)	1(1)	3(2)
3	4(4)	1(3)	3(1)	2(2)
6	4(4)	1(3)	3(1)	2(2)
9	4(4)	1(3)	2(1)	3(2)
12	4(4)	1(3)	3(1)	2(2)
15	4(4)	1(3)	3(1)	2(2)
18	4(4)	2(4)	3(1)	1(2)
21	4(4)	2(3)	3(2)	1(1)
24	4(4)	2(3)	3(2)	1(1)
28	4(4)	2(3)	3(2)	1(1)
32	4(4)	2(3)	3(2)	1(1)
36	4(4)	2(2)	3(3)	1(1)
40	4(4)	2(2)	3(3)	1(1)

Table 8 Results of **ExpCard** consider also the index cost for $|D_i| = 28$ splits.

#	DJNI	DJGI +2in	DJRE +1in	SJMR
1	2(2)	4(4)	3	1(1)
3	2(3)	4(4)	3(2)	1(1)
6	4(4)	3(3)	2(2)	1(1)
9	4(4)	3(3)	2(2)	1(1)
12	4(4)	3(3)	2(2)	1(1)
15	4(4)	3(3)	2(2)	1(1)
18	4(4)	3(3)	2(2)	1(1)
21	4(4)	3(3)	2(2)	1(1)
24	4(4)	3(3)	2(2)	1(1)
28	4(4)	3(3)	2(2)	1(1)
32	4(4)	3(3)	2(2)	1(1)
36	4(4)	3(3)	2(2)	1(1)
40	4(4)	3(3)	2(2)	1(1)

is the best choice. Indeed, the experimental evaluation revels that the estimate for DJGI is a bit conservative and this may produce an over-estimation of its cost. This is due to the fact that DJGI is the only algorithm that can have unbalanced maps even when applied to uniformly distributed datasets. Indeed, while for DJRE and SJMR the cell dimensions are always the same for the two datasets, with DJGI the two indexes can have different cells and the filtering phase can produce intersections with very different areas. As mentioned in Estimate 5, the filter factor ψ captures an average behaviour, and this can produce an over-estimation in the parallel execution cost, since we have no control about the scheduling of heavy and light maps inside the same node.

Tab. 9 reports the results of **ExpCard** without and with considering the index construction. In particular, column **G** is the cardinality in MBytes of the first dataset D_i, for each of them the cardinality of the second dataset D_j is changed from 128 MBytes (1 split) to 5120 MBytes (40 splits) and the averages are computed. Column $\mathbf{b} \in \mathbf{F}$ is the percentage of cases in which the operator b, which is the best in the experiments, is contained in the set F of non-dominated solutions. Similarly, as regards to the cumulative cost, column $\mathbf{b = f/s}$ contains the percentage of cases in which its suggestion f corresponds to the best operator b or the second one s, while **w=l** reports the percentage of cases in which the operator w, which is the worst in the experiments, is equal to the last algorithm for the cumulative cost. Column **%DL** is the average percentage of delay w.r.t b obtained by selecting the operator f (the lower the better), while columns **%GW** and **%GR** are the average percentage of gain

Table 9 Results of **ExpCard**. Column "G" is the cardinality in MBytes of D_i, for each of them the cardinality of D_j is changed from 1 to 40 splits.

G	**without index construction**						
	$b \in F$	$b = f$	$b = s$	$w = l$	%DL	%GW	%GR
128	92%	85%	8%	100%	5%	92%	36%
1152	100%	62%	0%	100%	9%	72%	45%
2304	100%	85%	0%	100%	6%	76%	51%
3584	100%	62%	8%	100%	7%	79%	55%
5120	100%	62%	23%	100%	9%	82%	59%
G	**with index construction**						
	$b \in F$	$b = f$	$b = s$	$w = l$	%DL	%GW	%GR
128	100%	38%	62%	100%	17%	56%	36%
1152	100%	85%	8%	100%	7%	69%	51%
2304	100%	100%	0%	100%	0%	78%	60%
3584	92%	100%	0%	100%	0%	81%	64%
5120	100%	100%	0%	100%	0%	83%	66%

in choosing f w.r.t. the experimental worst operator and a randomly chosen operator (the higher the better), respectively.

Experiment 2 (MBR size – ExpMBR)**.** Given the datasets considered in Exp. 1, **ExpMBR** changes the MBR size of each geometry to 1e−7 w.r.t. the reference space.

Tab. 10 reports the results of **ExpMBR** without and with considering the cost of the index construction, the column meanings is the same of **ExpCard**. Detailed results about each specific case can be found in [4].

Table 10 Results of **ExpMBR**. Column "G" is the cardinality in MBytes of D_i, for each of them the cardinality of D_j is changed from 1 to 40 splits. "IC" stands for index construction.

G	**without index construction**						
	$b \in F$	$b = f$	$b = s$	$w = l$	%DL	%GW	%GR
128	100%	92%	8%	100%	9%	48%	34%
1152	100%	54%	23%	100%	13%	68%	39%
2304	100%	92%	0%	100%	1%	77%	51%
3584	100%	69%	31%	100%	13%	78%	52%
5120	85%	54%	46%	100%	18%	80%	49%
G	**with index construction**						
	$b \in F$	$b = f$	$b = s$	$w = l$	%DL	%GW	%GR
128	100%	92%	8%	100%	8%	53%	37%
1152	100%	92%	8%	100%	0%	72%	53%
2304	100%	100%	0%	100%	0%	77%	58%
3584	100%	100$	0%	100%	0%	80%	62%
5120	100%	100%	0%	92%	0%	81%	62%

Experiment 3 (Overlapping – ExpOver)**.** Given the datasets considered in Exp. 1, **ExpOver** changes the percentage of overlapping of the two datasets to 50% and 25%.

Tab. 11 reports the results of **ExpOver** without and with considering the cost of the index construction and applying an overlap of 50% and 25% (column **OV**), the meaning of other columns is the same of **ExpCard**. Detailed results about each specific case can be found in [4].

Table 11 Results of **ExpOV**. Column "G" is the cardinality in MBytes of D_i, for each of them the cardinality of D_j is changed from 1 to 40 splits. "IC" stands for index construction.

G	**OV**	**without index construction**						
		$b \in F$	$b = f$	$b = s$	$w = l$	%DL	%GW	%GR
128	0.50	92%	85%	8%	100%	23%	55%	45%
1152	0.50	100%	54%	38%	100%	26%	76%	53%
2304	0.50	100%	85%	8%	100%	10%	84%	64%
3584	0.50	100%	62%	15%	100%	14%	85%	64%
5120	0.50	100%	69%	23%	100%	12%	86%	66%
128	0.25	92%	92%	0%	100%	4%	72%	54%
1152	0.25	100%	69%	31%	100%	53%	85%	67%
2304	0.25	100%	69%	31%	100%	33%	88%	71%
3584	0.25	100%	92%	8%	100%	6%	91%	79%
5120	0.25	100%	62%	38%	100%	37%	85%	63%
G	**OV**	**with index construction**						
		$b \in F$	$b = f$	$b = s$	$w = l$	%DL	%GW	%GR
128	0.50	100%	15%	85%	100%	36%	52%	32%
1152	0.50	100%	85%	0%	100%	9%	65%	46%
2304	0.50	100%	92%	8%	100%	2%	75%	56%
3584	0.50	100%	92%	8%	100%	2%	79%	60%
5120	0.50	100%	92%	8%	100%	1%	81%	63%
128	0.25	100%	15%	77%	100%	62%	46%	25%
1152	0.25	100%	85%	0%	100%	13%	62%	42%
2304	0.25	100%	92%	8%	100%	4%	73%	52%
3584	0.25	100%	92%	0%	92%	2%	78%	55%
5120	0.25	100%	92%	0%	100%	2%	79%	61%

These experiments confirm the good behaviour of the proposed cost model, which is able to detect the best algorithm in 88% of cases and it always gains from a minimum of 21% to a maximum of 67% w.r.t. a random choice.

7 Extension to Skewed Distributions

This paper concentrates on datasets which present a uniform distribution of their geometries. This assumption has two main effects: the first one is keeping the various formulas simpler and the second one is the possibility to concentrate on only one kind of partitioning technique, or global index. Anyway, the overall structure of the cost model and the majority of the estimation formulas do not depend on the dataset distribution and can be applied to all kinds of datasets. In this section, we want to take a look about which estimation formulas are general and which ones require some adaptation in order to be applicable in the general case.

The first formula that requires to be adapted in case of skewed datasets is certainly the selectivity estimation presented in Eq. 1 of Sect. 3. As done for the uniform case, some previous studies about selectivity estimation [29, 8] can be considered and adapted for using them in a MapReduce context, namely by adjusting the notion of reference area.

A second aspect to be considered is the choice of the global index. As we already mentioned, in case of skewed datasets, such choice can have a great impact on the overall spatial join performances. Moreover, some works have been done in order to determine the best partitioning technique to apply based on the dataset distribution [5]. Regardless of the kind of considered index, the application of a partitioning technique requires the definition of a grid, namely a set of cells on which geometries will be accommodated. What differentiates one index from another is the way such cells will be produced and their resulting shape and dimension. Notice that the aim of the more sophisticated partitioning technique is to produce balanced splits even in presence of non-uniformly distributed datasets. In other words, the shape and dimensions of such cells will be determined so that the number of geometries inside each cell is quite the same, this is achieved for instance by enlarging or shrinking cells with respect to the ones produced by a uniform subdivision of the dataset MBR. From this considerations, it follows that the estimated number of geometries per split ($\#geo_{\mathtt{sp}}$) can be estimated as in Eq. 6. Conversely, the estimation of the number of generated cells ($\#cells$) has to be appropriately adapted w.r.t. to the one proposed in Eq. 7. All the other estimation formulas in Sect. 4 can be applied also in presence of other partitioning techniques.

Relatively to the spatial join algorithms, great importance is covered by the cost of the plain sweep algorithm presented in Estimation 3. The general structure of this formula can be applied also to skewed datasets, but it depends on the selectivity estimation which has to be adapted as previously discussed. This holds also for other estimation formulas defined for the various spatial join algorithms which have a direct or indirect dependency to the selectivity. In case of the DJNI algorithm, since its geometry subdivision does not depend on spatial properties, the estimation formulas are all valid, except for the estimation of the number of pairs produced by each map ($joinSz_{\mathtt{map}}$) reported in Eq. 10 which depends on the selectivity. In case of the DJGI, we can assume that a more sophisticated partitioning technique is applied, namely one that takes into consideration the skewed nature of the dataset distribution. Therefore, as discussed above, each cell will contain an equal amount of geometries ($\#geo_{\mathtt{sp}}$), while a different number of cells ($\#cells$) and cell dimensions can be produced (len_x^{avg} and len_y^{avg}).

The situation could become worse in case of the DJRE algorithm, because as already mentioned, the choice of the best partitioning technique will depend on the specific dataset distribution. Therefore, if two datasets with really different distributions have to be joined, the choice to use the index of one dataset to repartition the other one can produce very unbalanced compound splits. Moreover, the assumptions made about the number of geometries per split

could not hold for the repartitioned dataset. Therefore, for this dataset also the estimation $\#geo_{\mathtt{sp}}$ has to be properly extended.

Finally, the SJMR can be considered the algorithm most disadvantaged by the presence of skewed dataset, because it always applies a uniform subdivision of the space. In this case, the estimations that require to be adapted is the number of geometries per cell, that could be very different from one cell to another, while the estimation of the number of cells remain the same. Finally, the selectivity estimation used in the various other formulas have to be properly extended for capturing the average, best and worst case scenario.

From all these considerations, it is clear that the general structure of the cost model and the majority of the defined estimation formulas are applicable also in presence of skewed datasets.

7.1 Experiments on skewed datasets

The last set of experiments we have performed involves real-world datasets, that are skewed in nature. In particular, we consider again the case studies described in the motivating example of Sect. 1 and we compare the suggestions produced by the proposed cost model with the results reported in Tab. 1. In addition we use two different clusters with different characteristics. Tab.12 describes both the datasets and the clusters characteristics.

Table 12 Real-world datasets. (D_{USAWA} denotes the water areas of USA, D_{USAPR} the primary roads of USA, D_{AUSTATES} the states of Australia and D_{AUROADS} the roads of the Australian network. From the second column, the size, cardinality of each dataset, the average area of the MBR of their geometries, and the average number of vertices of their geometries are shown. As regards to the cluster characteristics, the number of nodes and the number of containers for each nodes are reported.

Dataset					**Cluster**	
Name	**size**	**#geo**	**AVG MBR(geo)**	**AVG #Vert(geo)**	**#nodes**	**#containers per node**
D_{USAWA}	2.07 GB	2,281,276	1.8e-8	42.88	3	2
D_{USAPR}	0.91 GB	12,393	3.4e-6	2,089.7		
D_{AUSTATES}	18.3 MB	8	9.2e-2	61,126	10	1
D_{AUROADS}	146 MB	335,701	4.74e-7	14.57		

The proposed cost model has been applied to estimate the cost of the join operators in the two configurations. The results are summarized in Tab.13: column $\mathbf{b} \in \mathbf{F}$ reports if b, which is the best operator in the experiments, is contained in the set F of non-dominated solutions. Similarly, as regards to the cumulative cost, column $\mathbf{b = f/s}$ reports if the suggestion f corresponds to the best operator b or to the second one s, while $\mathbf{w{=}l}$ reports if the operator w, which is the worst in the experiments, is equal to the last algorithm for the cumulative cost. Column **%DL** is the average percentage of delay w.r.t b obtained by selecting the operator f (the lower the better), while columns

%GW and **%GR** are the average percentage of gain in choosing f w.r.t. the experimental worst operator and a randomly chosen operator (the higher the better), respectively. Since a different cluster has been used in each situation, some micro-benchmarks on each of them have been preliminary performed in order to obtain the weights to be used in the total cost estimation. From the obtained results, we can notice that:

- In the first case, the two datasets not only have different sizes (the first one is twice the size of the second one), but they also present very different spatial characteristics, like the number of vertices. Moreover, the datasets are not uniformly distributed. Therefore, the application conditions are far from the ideal hypothesis stated for the applicability of the cost model. However, the cost model is able to detect the best algorithm and also the gain with respect to the random choice is very good, about 52%. Conversely, the last predicted algorithm is not the last one in the experiments.
- In the second case, the differences between the two datasets are exhacerbated, they are tremendously different with respect to all parameters. In this case the cost model still behaves well, even if the conditions are far from the hypothesis. Indeed, the other positions in the ranking produced by the cost model are not consistent with the experiments.

Table 13 Results of experiments on real datasets. Columns "DS_1" ("DS_2") is the first (second) dataset name (D_{USAWA} denotes the water areas of USA, D_{USAPR} the primary roads of USA, D_{AUSTATES} the states of Australia and D_{AUROADS} the roads of the Australian network.

$\mathbf{DS}_1$	$\mathbf{DS}_2$	$b \in F$	$b = f$	$b = s$	$w = l$	%DL	%GW	%GR
D_{USAWA}	D_{USAPR}	yes	yes	no	no	0%	66.2%	52.6%
D_{AUSTATES}	D_{AUROADS}	yes	yes	no	no	0%	64.5%	78.7%

From the obtained results, we can essentially conclude that: the cost model is able to correctly capture and taking into consideration the spatial characteristics of the datasets, since it is able to work with very different datasets, in terms of cardinality, number of vertices and geometry extensions. Secondly, it works quite well also in presence of skewed distributions, but it has to be refined in order to better deal with this situations.

8 Conclusion

In this paper we present a cost model for ranking four MapReduce implementations of the spatial join. The cost model proposes for each algorithm some formulas for estimating the cost of the map, shuffle and reduce tasks distinguishing three components: CPU, Local I/O and Network I/O cost. The estimates vary according to the properties of the input datasets in terms of: cardinality, extent and number of vertices of the geometries, and the spatial overlapping of the datasets. Exhaustive experiments have been done using synthetic datasets with variable characteristic, considering both the dominance

relation and a cumulative cost obtained from the relations existing between the three cost components in the considered cluster. These experiments confirm the good behaviour of the proposed cost model. Future work will regard the extension of the cost model to other index types and dataset distributions on the basis of the considerations made in Sect. 7. Moreover, the cost model can be used as a basis for both implementing a spatial query optimizer for MapReduce frameworks and for improving the currently available partitioning techniques [3].

References

1. An, N., Yang, Z., Sivasubramaniam, A.: Selectivity estimation for spatial joins. In: Proceedings of the 17th International Conference on Data Engineering, pp. 368–375 (2001). DOI 10.1109/ICDE.2001.914849
2. Aref, W., Samet, H.: A cost model for query optimization using R-Trees. In: Proceedings of the Second ACM Workshop on Advances in Geographic Information Systems, pp. 60–67 (1994)
3. Belussi, A., Carra, D., Migliorini, S., Negri, M., Pelagatti, G.: What makes spatial data big? A discussion on how to partition spatial data. In: Proceedings of 10th International Conference on Geographic Information Science, pp. 1–15 (2018). DOI 10.1109/ICDE.2015.7113382
4. Belussi, A., Migliorini, S., Eldawy, A.: A Cost Model for Spatial Join Operations in SpatialHadoop. Tech. Rep. RR108/2018, Dept. of Computer Science, University of Verona (2018). URL https://iris.univr.it/handle/11562/981957
5. Belussi, A., Migliorini, S., Eldawy, A.: Detecting Skewness of Big Spatial Data in SpatialHadoop. In: Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 432–435 (2018). DOI 10.1145/3274895.3274923
6. van den Bercken, J., Seeger, B., Widmayer, P.: The bulk index join: a generic approach to processing non-equijoins. In: Proceedings of the 15th International Conference on Data Engineering, pp. 257– (1999). DOI 10.1109/ICDE.1999.754937
7. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A Comparison of Join Algorithms for Log Processing in MapReduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 975–986 (2010). DOI 10.1145/1807167.1807273
8. Chasparis, H., Eldawy, A.: Experimental Evaluation of Selectivity Estimation on Big Spatial Data. In: Proceedings of the Fourth International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data (GeoRich 2017), collocated with ACM SIGMOD 2017, pp. 8:1–8:6. Chicago, IL (2017). DOI 10.1145/3080546.3080553. URL https://doi.org/10.1145/3080546.3080553
9. Dittrich, J., Seeger, B.: Data redundancy and duplicate detection in spatial join processing. In: D.B. Lomet, G. Weikum (eds.) Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000, pp. 535–546. IEEE Computer Society (2000). DOI 10.1109/ICDE.2000.839452. URL https://doi.org/10.1109/ICDE.2000.839452
10. Eldawy, A., Alarabi, L., Mokbel, M.F.: Spatial Partitioning Techniques in SpatialHadoop. Proceedings of the VLDB Endowment **8**(12), 1602–1605 (2015). DOI 10.14778/2824032.2824057
11. Eldawy, A., Mokbel, M.F.: SpatialHadoop: A MapReduce framework for spatial data. In: Proceedings of the 31st IEEE International Conference on Data Engineering, pp. 1352–1363 (2015). DOI 10.1109/ICDE.2015.7113382
12. Eldawy, A., Mokbel, M.F.: Spatial Join with Hadoop. In: Encyclopedia of GIS, pp. 2032–2036. Springer (2017). DOI 10.1007/978-3-319-17885-1_1570
13. Eldawy, A., Mokbel, M.F.: The Era of Big Spatial Data. Proceedings of the VLDB Endowment **10**(12), 19921995 (2017). DOI 10.14778/3137765.3137828

14. Eldawy, A., Sabek, I., Elganainy, M., Bakeer, A., Abdelmotaleb, A., Mokbel, M.F.: Sphinx: Empowering Impala for Efficient Execution of SQL Queries on Big Spatial Data. In: Proceedings of the 15th International Symposium on Advances in Spatial and Temporal Databases, pp. 65–83 (2017). DOI 10.1007/978-3-319-64367-0_4
15. Gu, J., Peng, S., Wang, X.S., Rao, W., Yang, M., Cao, Y.: Cost-Based Join Algorithm Selection in Hadoop. In: Proceedings of the 15th International Conference on Web Information Systems Engineering, pp. 246–261 (2014). DOI 10.1007/978-3-319-11746-1_18
16. Han, S., Choi, W., Muwafiq, R., Nah, Y.: Impact of Memory Size on Bigdata Processing based on Hadoop and Spark. In: Proceedings of the International Conference on Research in Adaptive and Convergent Systems (RACS), pp. 275–280 (2017). DOI 10.1145/3129676.3129688
17. Harada, L., Nakano, M., Kitsuregawa, M., Takagi, M.: Query Processing for Multi-Attribute Clustered Records. In: Proceedings of 16th International Conferece on Very Large Data Bases, pp. 59–70 (1990)
18. Hoel, E.G., Samet, H.: Benchmarking Spatial Join Operations with Spatial Output. In: Proceedings of the 21th International Conference on Very Large Data Bases, pp. 606–618 (1995)
19. Jacox, E.H., Samet, H.: Spatial Join Techniques. ACM Transactions on. Database Systems **32**(1) (2007). DOI 10.1145/1206049.1206056
20. Lin, X., Meng, Z., Xu, C., M., W.: A Practical Performance Model for Hadoop MapReduce. In: Proceeding of the 2012 IEEE International Conference on Cluster Computing Workshops, pp. 231–239 (2012). DOI 10.1109/ClusterW.2012.24
21. Mamoulis, N., Papadias, D.: Multiway Spatial Joins. ACM Transactions on Database Systems **26**(4), 424–475 (2001). DOI 10.1145/503099.503101
22. Mavridis, I., Karatza, H.: Performance Evaluation of Cloud-based Log File Analysis with Apache Hadoop and Apache Spark. Journal of Systems and Software **125**(C), 133–151 (2017). DOI 10.1016/j.jss.2016.11.037
23. Nievergelt, J., Hinterberger, H., Sevcik, K.C.: The grid file: An adaptable, symmetric multi-key file structure. In: Proceedings of 3rd Conference of the European Cooperation in Informatics – Trends in Information Processing Systems, pp. 236–251 (1981). DOI 10.1007/3-540-10885-8_45
24. Papadopoulos, A., Rigaux, P., Scholl, M.: A Performance Evaluation of Spatial Join Processing Strategies. In: Proceedings of 6th International Symposium on Advances in Spatial Databases, pp. 286–307 (1999). DOI 10.1007/3-540-48482-5_18
25. Patel, J.M., DeWitt, D.J.: Partition based spatial-merge join. SIGMOD Record **25**(2), 259–270 (1996). DOI 10.1145/235968.233338
26. Rigaux, P., Scholl, M., Voisard, A.: Spatial Databases with Application to GIS. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
27. Sabek, I., Mokbel, M.F.: On Spatial Joins in MapReduce. In: Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 21:1–21:10 (2017). DOI 10.1145/3139958.3139967
28. Samadi, Y., Zbakh, M., Tadonki, C.: Performance comparison between Hadoop and Spark frameworks using HiBenchbenchmarks. Concurrency and Computation: Practice and Experience **30**(12) (2018). DOI 10.1002/cpe.4367
29. Siddique, A.B., Eldawy, A., Hristidis, V.: Comparing Synopsis Techniques for Approximate Spatial Data Analysis. PVLDB **12**(11), 1583 – 1596 (2019). DOI 10.14778/3342263.3342635. URL http://www.vldb.org/pvldb/vol12/p1583-siddique.pdf
30. Sowell, B., Salles, M.V., Cao, T., Demers, A., Gehrke, J.: An Experimental Analysis of Iterated Spatial Joins in Main Memory. Proceedings of the VLDB Endowment **6**(14), 1882–1893 (2013). DOI 10.14778/2556549.2556570
31. Šidlauskas, D., Jensen, C.S.: Spatial Joins in Main Memory: Implementation Matters! Proceedings of the VLDB Endowment **8**(1), 97–100 (2014). DOI 10.14778/2735461.2735470
32. White, T.: Hadoop: The Definitive Guide, 4th edn. O'Reilly Media, Inc. (2015)
33. Xie, D., Li, F., Yao, B., Li, G., Chen, Z., Zhou, L., Guo, M.: Simba: spatial in-memory big data analysis. In: Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 86:1–86:4 (2016). DOI 10.1145/2996913.2996935

34. Yu, J., Wu, J., Sarwat, M.: 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, pp. 70:1–70:4 (2015). DOI 10.1145/2820783.2820860