IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 1, 2022, accepted February 18, 2022, date of publication March 8, 2022, date of current version March 21, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3157405

Enabling Efficient Distributed Spatial Join on Large Scale Vector-Raster Data Lakes

SEBASTIÁN VILLARROYA, JOSÉ R. R. VIQUEIRA, JOSÉ M. COTOS, AND JOSÉ A. TABOADA

COGRADE, Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain

Corresponding author: Sebastián Villarroya (s.villarroya@usc.es)

This work was supported in part by the Spanish Ministry of Science and Innovation through Storage and Processing of Massive Geospatial Data for Smart and Sustainable Urban Transport (MaGIST) under National Project PID2019-105221RB-C42, and in part by the Galician Government under Grant ED431B 2021/16.

ABSTRACT Both the increasing number of GPS-enabled mobile devices and the geographic crowd-sourcing initiatives, such as Open Street Map, are determinants for the large amount of vector spatial data that is currently being produced. On the other hand, the automatic generation of raster data by remote sensing devices and environmental modeling processes was always leading to very large datasets. Currently, huge data generation rates are reached by improved sensor observation systems and data processing infrastructures. As an example, the Sentinel Data Access System of the Copernicus Program of the European Space Agency (ESA) was publishing 38.71 TB of data per day during 2020. This paper shows how the assumption of a new spatial data model that includes multi-resolution parametric spatial data types, enables achieving an efficient implementation of a large scale distributed spatial analysis system for integrated vector-raster data lakes. In particular, the proposed implementation outperforms the state-of-the-art Spark-based spatial analysis systems by more than one order of magnitude during vector-raster spatial join evaluation.

INDEX TERMS Large-scale data analysis, spatial analytics, spatial data management, vector-raster data analysis.

I. INTRODUCTION

Two major types of spatial datasets exists, namely *vector* and *raster* datasets. Vector datasets contain data of spatial entities, including the vector geometries that represent their location and shape in space. Much research effort has been devoted to vector spatial data management, which leaded to mature and standardized spatial DBMS solutions [1], [2]. Raster datasets contain the spatial or spatio-temporal distribution of variables such as air temperature, elevation above sea level, population density, etc. In general, they have the form of large 2D, 3D or 4D arrays of numeric real data, therefore they do not fit well with traditional database technologies. Although some specific scientific array data management solutions already exist [3], [4], most applications still rely on specific scientific file formats and ad-hoc programming.

The amount of available vector spatial data is increasing exponentially, mainly due to the generalized use of GPS-enabled mobile devices and to the arising of geographic crowd-sourcing initiatives. Examples of huge datasets obtained from GPS-enabled location-based applications are the approximately 250 million geo-tagged tweets generated per day in 2020 and the approximately 700K taxi trips stored per day in the NYC TLC trip record dataset during 2019. Another source of vector spatial data are the geographic crowd-sourcing initiatives. An example of such initiatives is the Open Street Map project, which provides access to a spatial vector dataset of about 1 TB. Raster datasets were always very large, because they are generated by automatic means, including remote sensing devices and environmental modeling processes. However, the advances in the hardware of sensor observation systems and data processing infrastructures are given rise to the increase of the raster data generation rate up to unprecedented levels. As an example, the Sentinel Data Access System of the Copernicus Program of the European Space Agency (ESA) was publishing 38.71 TB of data per day during 2020. As another example, the National Oceanographic and Atmospheric Administration (NOAA) of the U.S. Department of Commerce generates tens of terabytes of data per day from satellites, radars, ships, weather models and other sources.

To cope with the processing of the above data deluge, the arising of modern large scale data storage and processing technologies has caused the emergence of a new architectural

The associate editor coordinating the review of this manuscript and approving it for publication was Vlad Diaconita.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

trend in the development of Business Intelligence infrastructures, called the Data Lake [5]. Contrary to what happens in traditional data warehouses, in a Data Lake the data is stored without the need of a predefined schema designed to give response to an available list of queries. Data Lakes are created by inserting all the available raw data, regardless of its type, format and semantics, being flexible enough to enable future analysis tasks which are still not witnessed. Data lakes are implemented with Big Data technologies, including distributed file systems like HDFS for data storage and large scale data processing technologies like Apache Hadoop [6], for batch query processing, and Apache Spark [7] for on-line analytics.

Large scale spatial analysis in spatial Data Lakes is currently achieved by spatial extensions of either Apache Hadoop [8], [9] or Apache Spark [10]–[16]. In general, the available data storage features may be extended with i) spatial data types for vector geometries (points, linestrings, polygons, etc.), ii) spatial partitioning methods and iii) spatial indexing techniques. Besides, the data processing engine may be extended with new spatial operations and spatially enabled query optimization strategies. From all the above solutions, only GeoTrellis [15] has been designed to support raster data, and none of them enables the efficient integration of vector and raster spatial data analysis. In particular, to support spatial joins between vector and raster data, raster data has to be recorded as point data (one point object for each raster cell). Although data storage may still be efficient due to the data compression facilities incorporated in current distributed columnar data storage formats like Apache Parquet [17], data processing may not leverage the sampling nature of raster data to devise more efficient algorithms for spatial operations.

In this paper, it is shown how the assumption of an already existing integrated vector-raster data model approach enables the efficient implementation of a large scale vector-raster spatial on-line data analysis system on top of Apache Spark. In particular, it is shown through exhaustive experimentation how specific optimizations enable achieving response times for vector-raster spatial joins that are more than an order of magnitude faster than those achieved by currently available Spark-based spatial analysis systems.

More specifically, the contributions of the present work are resumed as follows.

- An implementation of the multi-resolution parametric spatial data types, the data structures and the operations of the data model is undertaken, using Apache Parquet [17] as the base data storage technology and Apache Spark [7] as the underlying large scale on-line data processing framework.
- An optimization strategy for vector-raster spatial join is designed and implemented, which leverages the special nature of raster data to enable the use of equi-join, instead of the spatial theta-join required by other approaches. The use of efficient algorithms for equi-join (sort-merge join and hash join) in Spark makes the present solution much faster than others, that need to implement spatial indexed nested-loop joins.
- A comparison through experimentation of the performance of the evaluation of the spatial join between vector (polygon) and raster data structures was performed between state of the art spark-based spatial analysis solutions. Response time, shuffle reads and writes costs, and peak execution memory consumption were measured. The present solution outperforms the fastest state of the art implementations by more than an order or magnitude, with a peak memory consumption which is in the order of the smallest ones.

The remainder of this paper is organized as follows. Section II provides an overview of other pieces of related work. The design of the data types, data structures and operations of the adopted vector-raster integrated data model is outlined in Section III. Section IV describes the implementation of the data model structures and operations on top of Apache Parquet and Apache Spark. The design and implementation of the optimization strategy for vector-raster spatial join evaluation is explained in Section V. Section VI discusses the results of the experimental evaluation that compares the performance of the present and state of the art implementations. Section VII concludes the paper and outlines issues for future work.

II. RELATED WORK

Most of the research undertaken in the area of Spatial Data Management has been focusing the effort in the effective and efficient management of spatial entity datasets, where vector representations for entity geometries are always considered. As a result of the above research, mature spatial SQL-based DBMSs [1], [2] are currently available, which implement the spatial part of the ISO SQL/MM standard [18]. Further on research has lead to the incorporation of spatial data management in modern columnar DBMSs [19] and NoSQL systems [20], [21].

Regarding raster data, current applications use to adopt specific data storage formats and processing libraries. The emergence of efficient scientific array data management systems such as Rasdaman [3] and SciDB [4] has opened the possibility to enable general purpose declarative raster data analysis. However, although some of the above spatial DBMS already incorporate specific raster data storage features, and despite of the existence of specific array data managers, to the best of these authors knowledge, effective and efficient declarative integrated management of vector and raster data has not been achieved by any available implementation.

If we restrict to data modeling, the integrated management of relational and array data is the aim of the SciQL [22] approach. However, spatial semantics are not implicitly included in array data, and the model becomes complex due to the combination of relational with array structures. On the other hand, integrated multiresolution spatial semantics are incorporated in MAPAL (Mapping Analysis Language), defined in the scope of the design of SODA [23],

a framework for Spatial Observation Data Analysis. The authors do not report on any available MAPAL implementation that proofs the viability of the approach. The data model assumed by the present implementation is based on the MAPAL data model.

Regarding large scale spatial data analysis, many systems have been already implemented. In general, all of them extend some already existing high performance data processing framework, either Apache Hadoop [6] or Apache Spark [7]. Data storage structures may be extended with spatial data representations, spatial partitioning and spatial indexing, whereas the data processing engine may be extended with new spatial operations and spatially enabled optimization strategies.

Hadoop GIS [8] enables running large scale spatial queries on top of Hadoop. A more efficient approach is adopted by SpatialHadoop [9], which extends Hadoop with spatial features at both storage and MapReduce layers. SpatialHadoop has been extended to provide native support for spatio-temporal data [24].

The broad majority of the most recent approaches are based on Apache Spark [7]. A nice survey that includes a performance comparison of the most relevant implementations is available in [25]. GeoSpark [10] extends Spark Resilient Distributed Datasets (RDD) with spatial data types, spatial partitioning and spatial indexing. On top of the Spatial RDD layer, spatial query processing is implemented with specific algorithms for spatial range, join and KNN queries.

The SpatialSpark [11] prototype implements spatial join queries on Spark. Different geometric types are supported, including points and polygons. Various spatial partitioning alternatives are provided and spatial indexing is done using R-trees. Range queries and both spatial and distance joins are supported.

Magellan [12] achieves distributed spatial analytics by extending SparkSQL [26] with spatial data types, such as *points*, *linestrings* and *polygons*, and predicates such as *within* and *contains*. Spatial indexing and spatial query processing is supported through the use of z-order curves.

Spatial query operators such as spatial range, spatial kNN, spatial join and kNN join are provided by LocationSpark [13] as a spatial query API on top of Apache Spark. It incorporates advanced query scheduling features that enables efficient management of query skew (uneven distribution of queries in space). As in most implementations, spatial indexing is used globally for spatial partitioning (either Grid Files or Quadtrees may be chosen) and also locally inside each partition (either R-trees, Quadtrees or Grid Files). Finally, the recording of statistics on data access is used to cache in memory most frequently used data.

Simba [14] (Spatial In-Memory Big Data Analysis) extends SparkSQL [26] with spatial data types and functions. It supports spatial indexing, both at global and local level, and it implements a cost-based query optimizer for effective spatial query plan selection.

GeoTrellis [15] is a high performance geoprocessing engine and programming toolkit, aimed at providing support for high performance geoprocessing web services. Contrary to all the above implementations, GeoTrellis supports both vector and raster data, however, integrated analysis of both through vector-raster joins operations is not supported.

Large scale spatio-temporal data analysis on top of Spark is implemented by Stark [16]. The implementation includes spatio-temporal operators for filter and join with different predicates, a kNN search operator and spatial partitioning and indexing.

In summary, many systems have been implemented to support the efficient spatial analysis of vector spatial data, including spatial DBMSs [1], [2], [19], NoSQL systems [20], [21] and many high performance distributed solutions based on either Hadoop [8], [9] or Spark [10]–[16]. Besides, scientific array data management systems [3], [4] may be used to perform analysis over raster datasets. However, integrated and efficient analysis of very large vector-raster datasets, through the support of vector-raster spatial joins, is not provided by any available solution.

III. DATA STRUCTURES AND OPERATIONS

The integrated data model for vector and raster data follows the approach proposed in [23] for heterogeneous spatio-temporal observation data. Data types, structures and operations are formalized below.

A. DATA TYPES

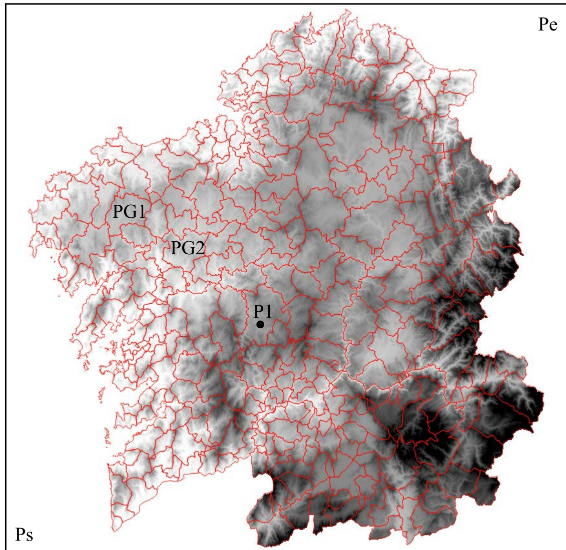
Besides the conventional data types typically supported by any data management system, the proposed model incorporates multiresolution parametric 2D spatial data types, which consist of fixed precision versions of those already proposed by the spatial part of the ISO SQL/MM standard [18]. If P (Precision) and R (Resolution) are two integer numbers ($P, R \in \mathbb{Z}$), then amongst other, the following two parametric spatial data types are incorporated ($\perp$ is used to denote the null value).

- *Point2D(P,R)*: $\{(x \cdot R, y \cdot R) \mid x, y \in \mathbb{Z} \wedge -10^P < x, y < 10^P\} \cup \{\perp\}$.
- *Polygon2D(P,R)*: Vector polygons whose borders are defined by sequences of elements of *Point2D(P,R)*.

It is noticed that data type *Polygon2D(P,R)* enables the integrated representation of both vector points and raster cells.

B. DATA STRUCTURES

Two data structures, namely *Dimensions* and *Extensional MappingSets*, enable the representation of both vector spatial entity sets and raster fields. A *Dimension* d over data type T, denoted $d : T$, is defined as a non-empty finite subset of $T - \{\perp\}$. *Dimensions* may be defined over any conventional data type, and also over data type *Point2D*, but not over geometric types like *Polygon2D*. Spatial *Samplings* are special cases of *Dimensions* of major interest for the present work.



Dimensions **Extensional MappingSets**

MunCode

...
15077
15078
...

Loc12m

start	end
Ps	Pe

Municipality

MunCode	Name	Geo
...	...	...
15077	Santa Comba	PG1
15078	Santiago	PG2
...	...	...

Topo

Loc12m	Elevation
Ps	null
...	...
P1	432.13
...	...
Pe	null

FIGURE 1. Illustration of spatial data structures.

Formally, if $s = (s_x, s_y)$ and $e = (e_x, e_y)$ are two elements of some type *Point2D(P,R)*, then a 2D *Sampling* W from s to e, denoted $W(s, e)$, is defined as the following *Dimension* over *Point2D(P,R)*.

$$W(s, e) = \{(x, y) \in Point2D(P, R) \mid s_x \leq x \leq e_x \wedge s_y \leq y \leq e_y\}$$

An *Extensional MappingSet* is a finite set of mappings (*Extensional Mappings*), with a shared domain, defined by the Cartesian product of *Dimensions*. Formally, if $d_1, d_2, \ldots, d_n$ is a sequence of *Dimensions* and T is a data type, then an *Extensional Mapping* with signature $M(d_1, d_2, \ldots, d_n) : T$ is defined as a mapping $M : d_1, d_2, \ldots, d_n \rightarrow T$. Based on the above, an *Extensional MappingSet* with signature $EM(d_1, d_2, \ldots, d_n \mid M_1 : T1, M_2 : T2, \ldots, M_m : Tm)$ is defined as the following set of mappings:

$$\{M_1(d_1, d_2, \ldots, d_n) : T_1, M_2(d_1, d_2, \ldots, d_n) : T_2, \ldots, M_m(d_1, d_2, \ldots, d_n) : T_m\}.$$

Spatial vector entity sets are represented in the model following a functional database approach [27]. Thus, Fig. 1 illustrates the representation of a collection of municipalities within an *Extensional MappingSet* with signature

$$Municipality(MunCode \mid Name : CString, Geo : Polygon2D(9, 10)),$$

where *MunCode* is a *Dimension* over data type *Integer*, that records municipality identification codes. *Extensional Mappings Name* and *Geo* yield respectively the name and geometry of the municipality corresponding to each identification code. Regarding raster fields, they are elegantly represented by *Extensional Mappings* whose domain is a 2D *Sampling*. Thus, *Extensional MappingSet* $Topo(Loc12m|Elevation : Real)$ in Fig. 1 represents a raster field of elevation above sea level. The bounds and resolution of raster domain are recorded in the 2D *Sampling*

$$Loc12m(Ps : Point2D(6, 12), Pe : Point2D(6, 12)).$$

It is important to remark that, in general, the values of a *Dimension* have to be explicitly recorded in order to represent it, as it is the case of *Dimension MunCode* above. However, in the case of a 2D *Sampling* such as *Loc*12*m* above, it is enough to store the bound values (*Ps* and *Pe* in the example), since all the other may be automatically generated.

C. OPERATIONS

The set of operations on *Dimensions* and *Extensional Mappings* that enable integrated vector-raster spatial analysis is introduced in this subsection.

1) *DIMENSION* OPERATIONS

They enable scanning *Dimensions* from storage, generate 2D *Samplings* from constants, performing set operations between *Dimensions*, and obtaining *Dimensions* from *Extensional MappingSet* projections.

- *ScanDimension*[*name*]. Scans the *Dimension* called *name* from the storage.
- *SamplingDimension*[*name*](K_1, K_2). Generates a new 2D *Sampling* *name*(K_1, K_2) using as bounds the values of *Constants* K_1 and K_2.
- *Union*(D_1, D_2). If neither d_1 nor d_2 is a 2D *Sampling* then it obtains a new *Dimension* by performing the set union between *Dimensions* d_1 and d_2. If either of d_1, d_2 is a 2D *Sampling*, then the result is the 2D *Sampling* with minimum extension that contains both d_1 and d_2.
- *Intersection*(D_1, D_2). Obtains a new *Dimension* by performing the set intersection between *Dimensions* D_1 and D_2. Contrary to the case of operation *Union*, the result is a 2D *Sampling* only if both D_1 and D_2 are 2D *Samplings*.
- *ProjectDimension*[s][c](MS). Operand *MS* is an *Extensional MappingSet*, parameter s is a reference to either a *Dimension* or a *Extensional Mapping* of *MS*, and parameter c is a reference to an *Extensional Mapping* of *MS* of a *Boolean* type. The result is a new *Dimension* containing all the values of s where c has true value.

2) *EXTENSIONAL MappingSets* OPERATIONS

They generate new *Extensional MappingSets* from existing *Dimensions*, *Extensional MappingSets* and *Constants*.

- *Product*[*name*]$(D_1, D_2, \ldots, D_n)$. Generates a new *Extensional MappingSet*, without *Extensional*

Mappings, whose domain is the Cartesian product $D_1 \times D_2 \times \ldots \times D_n$.

- $Product(MS, D)$. The domain of the result *Extensional MappingSet* is extended to the Cartesian product between the domain of MS and D.
- $ProjectMappingSet[s_1, s_2, \ldots, s_n](MS)$. The result *Extensional MappingSet* has the same domain of MS, but only the mappings of MS referenced by $s_1, \ldots, s_n$.
- $IMappings[m_1, m_2, \ldots, m_n](MS)$. Enables the evaluation of primitive intensional mappings over the *Extensional Mappings* and *Dimensions* of MS. Each m_i in the list of parameters is an expression of the form

$$M = pm(s_1, s_2, \ldots, s_m)$$

where M is the name for a new *Extensional Mapping* that will be added to MS, pm is a primitive mapping supported by the systems and each s_i is a reference to either a *Dimension* or an *Extensional Mapping* of MS.

- $EMapping[m](MS1, MS2)$. Enables the evaluation of an *Extensional Mapping* of $MS2$ over the *Dimensions* and *Extensional Mappings* of $MS1$. The expression m has the form

$$M = em(s_1, s_2, \ldots, s_n)$$

where M is the name of a new *Extensional Mapping* that will be added to MS, em is the name of an *Extensional Mapping* of $MS2$ and each s_i is a reference to either a *Dimension* or *Extensional Mapping* of $MS1$. To be able to do the evaluation, the domain of $MS2$ must be composed of a sequence of *Dimensions* $D_1 \times D_2 \times \ldots \times D_n$ such that the data type of each D_i is compatible with the data type of the relevant s_i.

- $KMapping[name](MS, K)$. Appends a new *Extensional Mapping* called *name* to MS with a *Constant* value K.
- $AggMapping[gb][ob][c][ag_1, ag_2, \ldots, ag_n](MS)$. Generates a new *Extensional MappingSet* by computing aggregates over part of the domain of MS. Parameter gb is a sequence of *Dimensions* strictly contained in the domain of MS. Parameter ob is an *order by specification* expression composed of pairs (s, o), where s references either a *Dimension* or *Extensional Mapping* of MS and o is an ordering direction, either *Ascending* or *Descending*. Parameter c is an *Extensional Mapping* of MS of a Boolean data type. Each ag_i is an expression of the form

$$M = aggregateMapping(s_1, s_2, \ldots, s_m)$$

where *aggregateMapping* is a primitive aggregate mapping supported by the system, such as *sum*, *avg*, *count*, *rank*, etc. and each s_i is a reference to either a *Dimension* or *Extensional Mapping* of MS. The domain of the result *Extensional MappingSet* is defined by gb *Dimensions*, and it has an *Extensional Mapping* recording the result of each ag_i expression. The aggregate mapping is evaluated only over elements where c is true. The *order by* specification ob is optional and required for some aggregate Mappings like *rank*.

Dimensions

Name	DataType	Size	Sampling	Start	End	FilePath
Loc12m	Point2D(7,12)	298472418	true	207406379667716	207759539666959	/pathToFile/...
MunCode	CString	314	false			

MappingSets

Name	Dimensions	FilePath
Topo	{Loc12m}	/pathToFile/...
Municipality	{MunCode}	/pathToFile/...

Mappings

MappingSet	Mapping	DataType
Topo	Elevation	FixedPrecision(7,3)
Municipality	Name	CString
Municipality	Geo	Polygon(9, 0.01)

FIGURE 2. Catalog example.

Complex spatial analysis tasks may be expressed with combinations of the above operations. A practical example that will be used to describe important optimizations is given below, which uses the data shown in Fig. 1 to obtain the average of elevation inside each municipality.

1) $D_1 = ScanDimension(MunCode)$
2) $D_2 = ScanDimension(Loc12m)$
3) $MS_1 = Product(D_1, D_2)$
4) $MS_2 = EMapping[Geo = Geo(MunCode)]$ $(MS_1, Municipality)$
5) $MS_3 = IMapping[c = contains(Geo, Loc12m)]$ (MS_2)
6) $MS_4 = EMapping[elev = Elevation(Loc12m)]$ $(MS_3, Topo)$
7) $MS_5 = AggMapping[MunCode][][c]$ $[avgElev = avg(elev)](Ms_4)$

IV. DISTRIBUTED IMPLEMENTATION

An implementation of the data structures and operations introduced in Section III in a distributed large scale data processing platform is described in the following subsections. The platform used is based on the combination of the column-oriented distributed Apache Parquet [17] data format with the large scale data analysis engine Apache Spark [7].

A. DATA STRUCTURES IMPLEMENTATION

Efficient structures to record both data and metadata of *Dimensions* and *Extensional MappingSets* have been implemented. Structures for both primary (in-memory) and secondary (disk) storage are described in the following subsections. To enable the recording of values of the spatial data types in Spark and Parquet, relevant Spark User Defined Types (UDT) where first implemented.

1) DISK STRUCTURES

A *Catalog* recording relevant metadata of *Dimensions* and *Extensional MappingSets* is stored in disk and loaded in main memory on system startup. Fig. 2 illustrates the contents of the *Catalog* with the metadata corresponding to the example of municipalities and elevation data of Fig. 1.

Besides the name, data type and size, stored metadata for *Dimensions* includes a boolean property that specifies whether the *Dimension* is a *2D Sampling* or not. The values of the bounds (*start* and *end* value) of *2D Samplings* are directly recorded in the *Catalog*. On the other hand, for non-sampling *Dimensions*, the path to the file with the data is required instead. For each *Extensional MappingSet*, the *Catalog* records its name, a reference to each *Dimension* of its domain and the path to the data file. The name and data type of each *Extensional Mapping* is also recorded in the *Catalog*.

The data of each non-sampling *Dimension* and each *Extensional MappingSet* is stored in a Parquet file. A *Dimension* Parquet file has two columns, one to record the actual *Dimension* values and another one that records an automatically generated reference of a long integer type. This reference column is used to enable late materialization [28] of *Dimension* values, as it will become clear later.

Regarding *Extensional MappingSet* data, the relevant Parquet file contains one column of the appropriate data type per *Extensional Mapping* plus an additional reference column. Each different value of the reference column is actually referencing a combination of values of the *Dimensions* of the *Extensional MappingSet* domain. More precisely, if $D_1 \times D_2, \times \ldots \times D_n$ is the domain of an *Extensional MappingSet* MS, then a reference RMS inside the *Extensional MappingSet* MS may be obtained from the references RD_i inside each specific *Dimension* D_i and vice-versa as follows.

$$RMS = \sum_i \left(RD_i * \prod_{j>i} size(D_j) \right)$$

$$RD_i = \left\lfloor \frac{RMS \ mod \prod_{j \geq i} size(D_j)}{\prod_{k>i} size(D_k)} \right\rfloor$$

Due to the above, combinations of the domain for which all the Extensional Mappings are undefined do not need to be recorded and late materialization [28] of *Extensional Mappings* is still enabled.

The compression techniques and appropriate encoding systems enabled by the Parquet storage format provides a drastic reduction of the storage payload. Besides, the use of fixed precision parametric spatial data types enables also the optimization of the storage for vector geometries, since integer values of appropriate sizes may now be used to store the coordinates, contrary to the double precision real values used by other approaches.

2) IN-MEMORY STRUCTURES

An appropriate structure, composed of data and metadata (header) areas, has been defined to record *Dimensions* and *Extensional MappingSets* in main memory. The header includes metadata such as name, size and data type. A boolean attribute *IsStored* is recorded in the header to identify whether the *Dimension* has been obtained form disk or generated in memory as a result of some operation. Attribute *StorageName* references the name of the *Dimension* stored

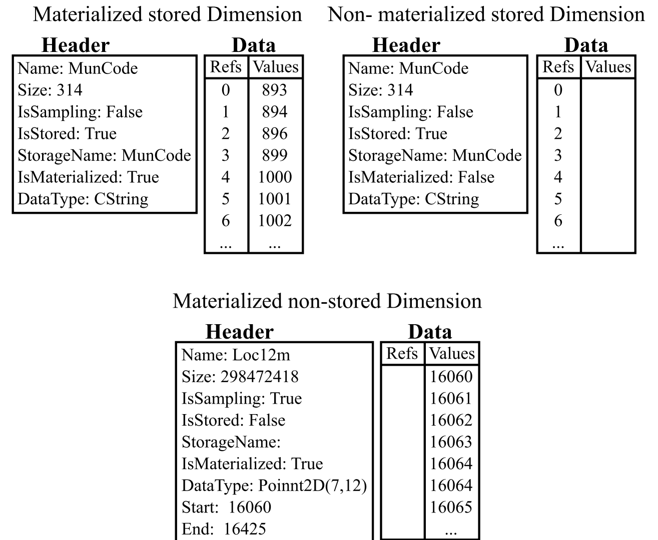


FIGURE 3. Illustration of in-memory ***Dimension*** structures.

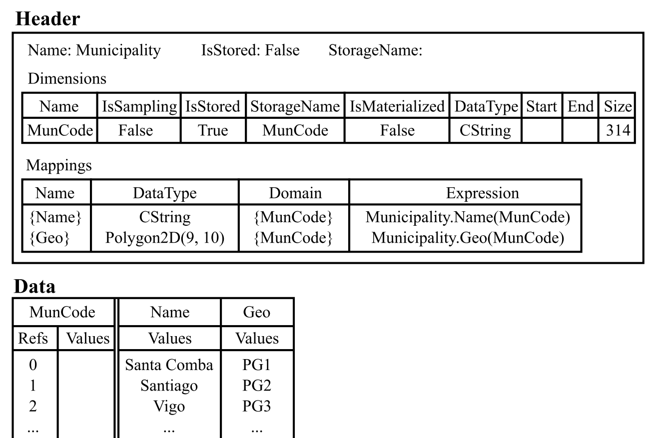


FIGURE 4. Illustration of in-memory ***Extensional MappingSet*** structures.

in the *Catalog*. A boolean attribute *IsMaterialized* is used to identify *Dimensions* materialized in main memory. *Dimensions* are materialized only when their specific values are needed for some computations [28]. Notice that the references of a *Dimension* may be generated in memory, as soon as they are required, from its size. The references and values of a Dimension are recorded in memory in a Spark *Dataframe*. Fig. 3 shows the header and DataFrame of three *Dimensions*, a materialized stored *Dimension*, a non-materialized stored *Dimension*, and a materialized non stored *Dimension*. Notice that a non materialized non stored *Dimension* has no sense.

Fig. 4 illustrates the in-memory metadata and data structures of the Municipality *Extensional MappingsSet* of Fig. 1. The header records metadata of both *Dimensions* and *Extensional Mappings*. Dimension metadata is the same to that recorded for isolated *Dimensions*, which was explained above. Regarding *Extensional Mappings*, the name and data type is recorded in the header. Besides, the expression that was used to compute the *Extensional Mapping* is also recorded, to avoid duplicate computations during query evaluation. If a new *Extensional Mapping* has to be computed

with an expression that is already in the header, the computation is avoided and the new name is simply added to the list of names of the *Extensional Mapping*. Finally, the list of Dimensions of the *Extensional MappingSet* domain from which the *Extensional Mapping* is directly dependent is also referenced. Notice that some computed *Extensional Mappings* might not depend on the whole domain, and this information is very useful during the evaluation of the *AggMapping* operation, as it will be explained in the following subsection.

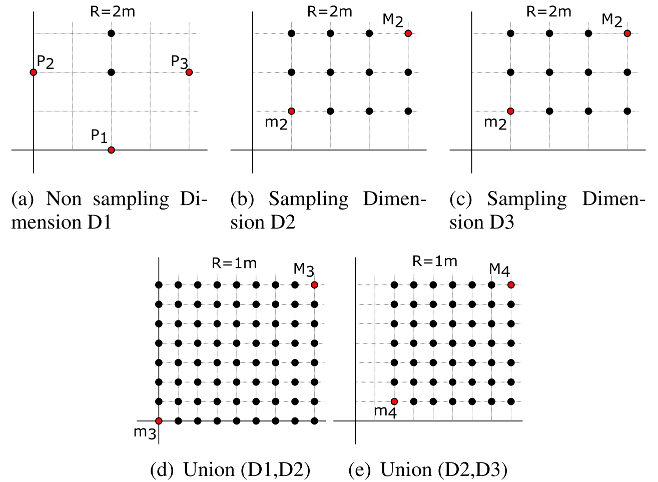
The data of each *Dimension* and *Extensional Mapping* of a given *Extensional MappingSet* is recorded in-memory in a Spark Dataframe. As in the case of isolated *Dimensions*, each *Dimension* of the domain may need two Dataframe columns, one to record the values and another one to record references. In the case of *Extensional Mappings*, one column is always needed to record the values, but additionally, if those values reference values already recorded in some *Dimension* then additional reference columns may also be recorded. The fact that one or various *Dimensions* are referenced by a specific *Extensional Mapping* must also be recorded in the header, to enable the interpretation of the relevant Dataframe columns.

B. OPERATIONS IMPLEMENTATION

Spark Dataframe operations are used to implement those operations defined in Subsection III-C. Implementation details are given below.

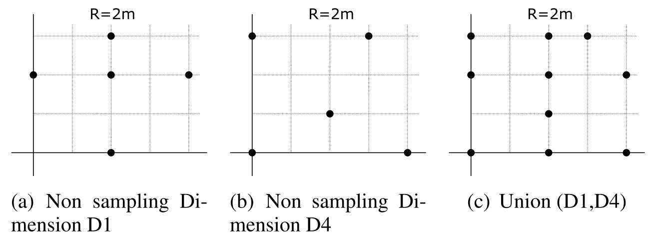
1) *DIMENSION* OPERATIONS

- *ScanDimension*[*name*]. Accesses the *Catalog* to locate the stored metadata for the *Dimension* called *name* and generates the references column in the in-memory structure, using the *Dimension* size and Spark operation *Range*.
- *SamplingDimension*[*name*](K_1, K_2). Generates an in-memory (non stored) materialized *2D Sampling*, called *name*, from the values of K_1 and K_2. The coordinates of the *2D Sampling Point2D* elements are generated by combining Spark operations *Range* and *Join* (equivalent to a Cartesian Product when called with no conditions), before they are recorded in the column of the relevant *Point2D* Spark UDT.
- *Union*(D_1, D_2). Always returns a materialized non stored *Dimension*. If at least one of the input *Dimensions* is a *2D Sampling*, then the result *2D Sampling* is computed from the minimum and maximum values of coordinates of elements recorded in those input *Dimensions*. Different scenarios for the evaluation of this operation with involved *2D Samplings* are shown in Fig. 5. If both *Dimensions* are non-sampling, then the operation is implemented using the operations *UnionAll* and *DropDuplicates* of Spark Dataframes. Fig. 6 illustrates this operation between two non sampling spatial *Dimensions*.
- *Intersection*(D_1, D_2). Again, it always returns a materialized non-stored *Dimension*. If any of the *Dimensions* is non-sampling, then the operation is implemented using *Dataframe* operation *Intersect*. These cases are illustrated in Fig. 7. On the other hand, if both *Dimensions* are *2D Samplings*, the bounds of the result are directly computed from the input bounds and next the result *2D Sampling* is generated as in operation *SamplingDimension*. This case is illustrated in Fig. 8.
- *ProjectDimension*[*s*][*c*](*MS*). This operation generates a non stored *Dimension*. To implemented it, first the *Filter* Dataframe operator is applied to the Dataframe of *MS* to evaluate condition *c* and discard relevant tuples. Next, Dataframe operation *Select* is used to project on the desired columns, referenced by *s*, and finally operation *DropDuplicates* is executed to eliminate duplicates from the result. Notice that duplicates have to be eliminated regardless of whether the *Dimension* is materialized or not.



(a) Non sampling Dimension D1 (b) Sampling Dimension D2 (c) Sampling Dimension D3
(d) Union (D1,D2) (e) Union (D2,D3)

FIGURE 5. Examples of operation Union with sampling Dimensions.

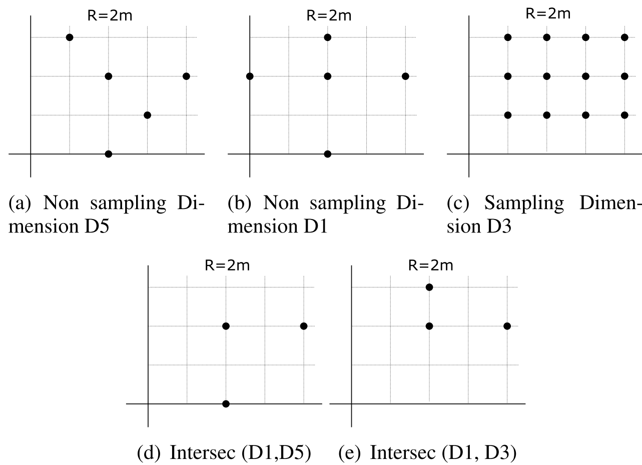


(a) Non sampling Dimension D1 (b) Non sampling Dimension D4 (c) Union (D1,D4)

FIGURE 6. Examples of operation Union with non sampling Dimensions.

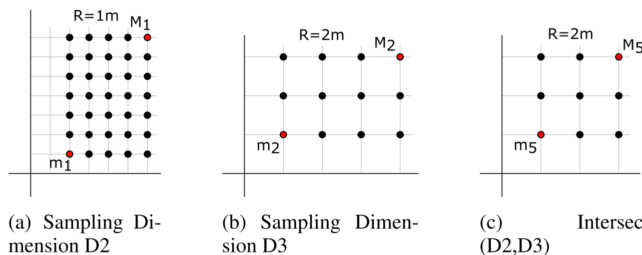
2) *EXTENSIONAL MappingSets* OPERATIONS

- *Product*[*name*]($D_1 \dots D_n$). The *Cartesian Product* of the input *Dimensions* is generated using the Dataframe *Join* operation.
- *Product*(*MS*, *D*). Again, the Dataframe operation *Join* is used to perform the *Cartesian product* between the Dataframes of *MS* and *D*.
- *ProjectMappingSet*[$s_1, \dots, s_n$](*MS*). Dataframe operation *Select* is here used to project from the input *MS* Dataframe the columns referenced by parameters s_i. Given that *Extensional Mappings* are always materialized, non-materialized *Dimensions* referenced



(a) Non sampling Dimension D5 (b) Non sampling Dimension D1 (c) Sampling Dimension D3
(d) Intersec (D1,D5) (e) Intersec (D1, D3)

FIGURE 7. **Examples of operation Intersection with non sampling Dimensions.**



(a) Sampling Dimension D2 (b) Sampling Dimension D3 (c) Intersec (D2,D3)

FIGURE 8. **Examples of operation Intersection with sampling Dimensions.**

by s_i must be materialized before applying the operator *Select*.

- $IMappings[m_1, m_2, \ldots, m_n](MS)$. Firstly, one SQL-like expression is generated for each m_i. Then, DataFrame operator *SelectExpr* can be used to evaluate them and generate one new column per *intensional* mapping expression. Notice that *primitive* mappings must be available as Spark User Defined Functions (UDFs) in order to be called by operator *SelectExpr*.
- $EMapping[m](MS1, MS2)$. Remember that input parameter m is an expression of the form

$$M = em(s_1, \cdots, s_m),$$

where em references an *Extensional Mapping* of $MS2$, and each s_i is the name of either a *Dimension* or an *Extensional Mapping* of $MS1$. If $D_1, \ldots, D_m$ are the *Dimensions* of the domain of $MS2$, the evaluation of em is performed with an equi-join operation between $MS1$ and $MS2$ where each column of references of each D_i in $MS2$ is equal to a relevant column of references of s_i in $MS1$. Such reference columns of $MS1$ have to be obtained from each *Dimension* D_i before the equi-join is performed.
- $KMapping[name](MS, K)$. A new column containing the same value in all rows is generated from K and passed as parameter to the Dataframe operator *WithColumn* to be added to the DataFrame of MS.
- $AggMapping[gb][ob][c][ag_1, ag_2, \ldots, ag_n](MS)$. Implementation of this method is as follows. First, the DataFrame operation *Filter* is applied to MS to discard element for which condition c do not hold. Next, DataFrame operation *Select* is used to drop from MS those *Dimensions* not referenced in input argument gb and those *Extensional Mappings* whose domain contains *Dimensions* not present in input argument gb (except those referenced by aggregated mappings). Notice that non-materialized *Dimensions* referenced by aggregated mappings must be materialized. Then, the DataFrame operator *GroupBy* prepares the DataFrame of MS for the subsequent application of relevant aggregate mappings by using DataFrame operator *Agg*. Appropriate aggregate mappings, implemented as UDFs, are used by this operator. Aggregate mappings that require an order by specification like *rank* are not supported yet.

V. VECTOR-RASTER SPATIAL JOIN OPTIMIZATION

The evaluation of *Cartesian Products* in *Extensional MappingSet* operation *Product*, involving either *Dimension* reference columns or raster points, makes the implementation described in the previous Section highly inefficient. However, the optimization of the data structures, with the incorporation of compact representations for reference ranges and raster geometries, enables the implementation of vector-raster spatial join efficiently, by avoiding the theta-joins that must be used in other approaches. This optimization is illustrated below with the help of the query example introduced in Subsection III-C.

It is first noticed that, actually, steps [1-5] of the example are performing a spatial join with spatial predicate *Contains* between municipality polygons and topo raster elements. In step (1), *Dimension* MunCode is scanned and therefore, a *Dataframe* with as many references as the size of the *Dimension* is generated (314 elements in this example). It is obvious that, those references are not needed until they are used to evaluate some *Extensional Mapping* of *Extensional MappingSet Municipality* (This will be done in step (4)). Therefore, the set of references could be replaced in the mean time by a compact range representation, composed of a couple of integers. The Dataframe recording such compact representation of references for *Dimension MunCode* is shown in Fig. 9 (a). The same optimization is applied also to the *ScanDimension* of step (2), whose compact result for *Dimension Loc12m* is depicted in Fig. 9 (b).

Operation *Product* in step (3) was very costly before the above described optimization, and it can now be applied without any problem, since each Dataframe contains just one row. The result Dataframe for *Extensional MappingSet* MS_1 is depicted in Fig. 9 (c), and as it may be observed in the figure, it maintains the compact range representation for *Dimension* references.

In step (4), the Geo *Extensional Mapping* of Municipality has to be evaluated for each MunCode of MS_1. To be able to perform this evaluation with operation *EMapping*, references are needed for *MunCode*. To obtain those references this operation must unnest the compact range representation of

Refs	Values
[0-314]	

(a) D_1

Refs	Values
[0-298472418]	

(b) D_2

MunCode		Loc12m	
Refs	Values	Refs	Values
[0-314]		[0-298472418]	

(c) MS_1

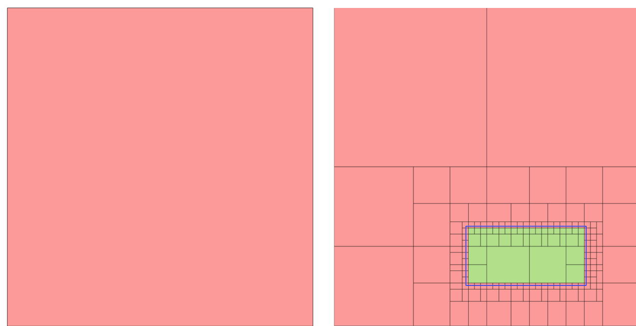
MunCode		Loc12m		Geo
Refs	Values	Refs	Values	Values
0		[0-298472418]		PG1
1		[0-298472418]		PG2
...		. . .		...
314		[0-298472418]		PG315

(d) MS_2

MunCode		Loc12m		Geo	C
Refs	Values	Refs	Values	Values	Values
0		[0-22678]	R11	PG1	True
0		[22678-232234]	R12	PG1	False
...		. . .	...	...	...
0		[122345-298472418]	R1n	PG1	True
1		[0-232234]	R21	PG2	False
...		...	...	...	...
1		[45452234-298472418]	R2n	PG2	True
...		...	...	...	...

(e) MS_3

FIGURE 9. Illustration of optimized intermediate structures.



(a) Initial rectangle (b) Resulting rectangles

FIGURE 10. Rectangles of the optimized *SpatialJoin* operator.

the references. Next, those references may be used to generate references for *Extensional MappingSet Municipality*, which will be used in an equi-join operation to attach the *Geo Extensional Mapping* to the *Dataframe* of MS_2. Such a Dataframe, with the unnested references column for *Muncode* and the *Geo* column is depicted in Fig. 9 (d).

The *Intensional Mapping Contains* that has to be evaluated by operation *IMapping* in step (5) needs materialized Municipality polygons and raster points. Municipality polygons were already obtained in the previous step, however, raster points are still not available in the input Dataframe. To minimize the number of rows generated by this operation, *Contains* has to be optimized to be applied between polygons

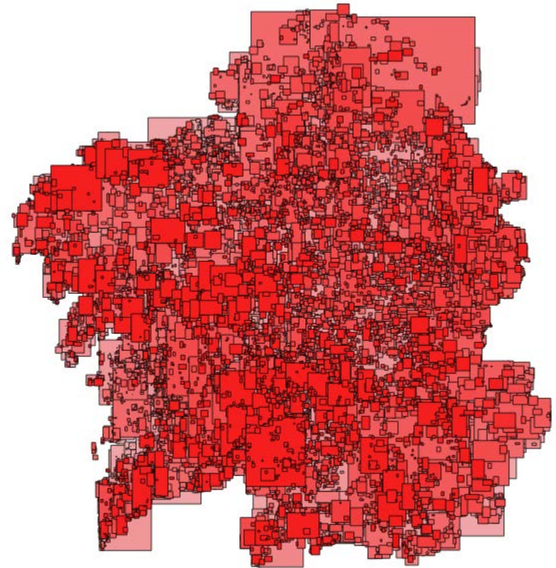


FIGURE 11. Combustion model MBRs.

and raster geometries (rectangles). To achieve this, each raster element in the Dataframe is decomposed in rectangles using the minimum bounding rectangle of each Municipality polygon. This rectangle decomposition, which is based on the regular decomposition of space followed by Quadtrees and z-curves is illustrated in Fig. 10. The decomposition process stops when one of the following condition holds: 1) the rectangle is inside the polygon, 2) the rectangle is outside the polygon, and 3) the rectangle size reaches the raster resolution. The rectangles that are inside the relevant polygon will have a True value in the new *Extensional Mapping c*. The Dataframe of the result *Extensional MappingSet*, which contains raster rectangles and *c* boolean values is depicted in Fig. 9 (e).

It is noticed that the data recorded in the Dataframe of *Extensional MappingSet* MS_3 of Fig. 9 (e) contains actually the spatial join between the raster points and the vector polygons, although raster points are represented in a compact format in the form of rectangles. It is also notice that up to this point, there system did not executed any costly Cartesian Product or theta-Join operation, and of course it did not needed any spatial indexed nested-loop join algorithm, which is the one commonly used in other approaches.

Finally, although the spatial join has already been performed, it is useless if the elevation data is not obtained from disk. This is done in step (6), where *Extensional Mapping* Evaluation has to be evaluated for each raster point of Loc12M in MS_3. Given that only points with *c* value True are used by the AggMapping operation in step (7), a last optimization consists in only obtaining from disk elevations for those points. To achieve this, first the rows with $C = True$ must be unnested to generate all the required Loc12m references. Those references are used in a subsequent equi-join operation to obtain the required elevation data from disk.

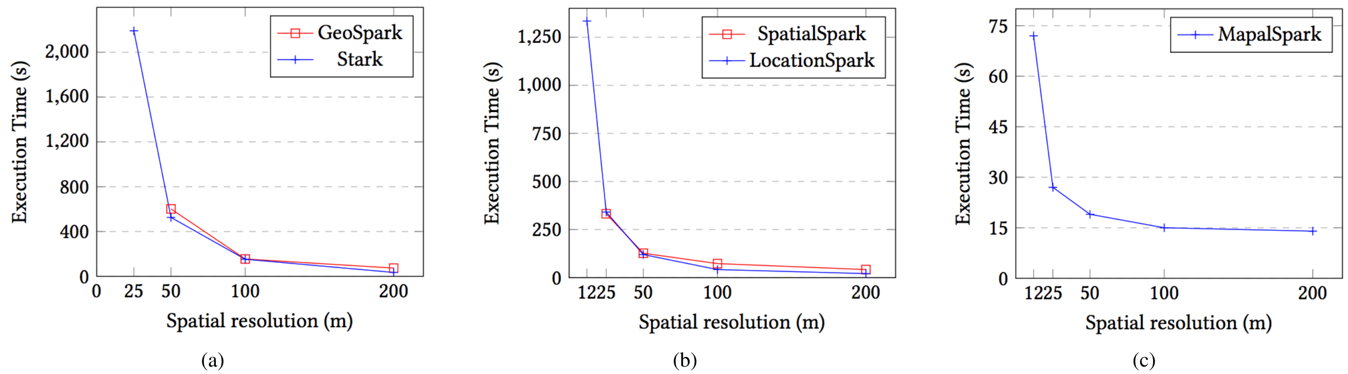


FIGURE 12. Spatial join runtime in a 40-executor cluster. (a) GeoSpark and Stark dataload results. (b) SpatialSpark and LocationSpark dataload results. (c) MapalSpark dataload results.

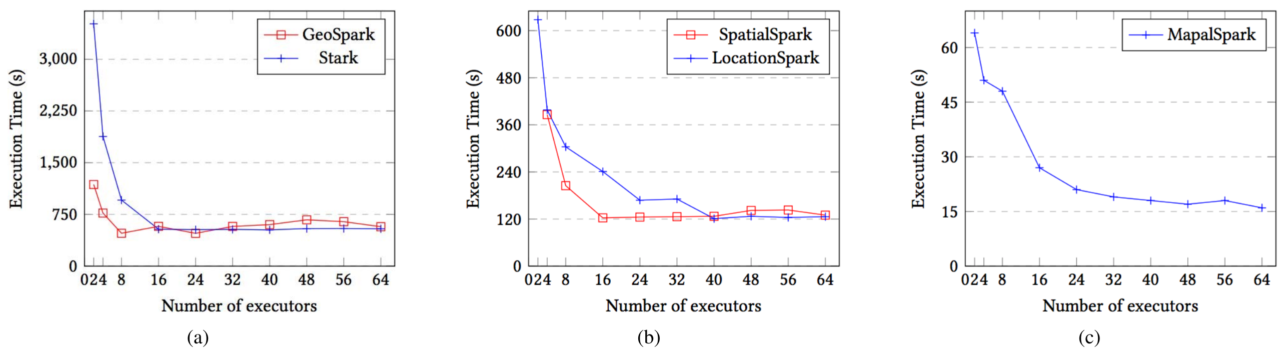


FIGURE 13. Spatial join scalability for a spatial resolution of 50 m. (a) GeoSpark and Stark scalability results. (b) SpatialSpark and LocationSpark scalability results. (c) MapalSpark scalability results.

The performance of this optimized set of operations is compared in the next section with the spatial join between municipality polygons and raster points that is supported by most of the available Spark-based spatial large scale analysis systems.

VI. EXPERIMENTAL EVALUATION

A. EXPERIMENTAL SETUP

This section provides an in-depth comparison between several state of the art solutions and the solution proposed in this work. In order to test the above-mentioned optimizations, the algorithm introduced in Subsection III-C has been implemented to obtain the mean elevation (raster data) in different regions (vector data) of Galicia. Since the number of municipalities in Galicia is not large, regions with different combustion properties have been used. Thus, the data analysis to be performed is "Obtain the mean elevation of every combustion region in Galicia".

To best of the author's knowledge, no standard benchmark has been developed to compare vector-raster data analysis operations or systems. Therefore, a number of experiments have been defined to provide a fair comparison between the available systems and the system developed in this work.

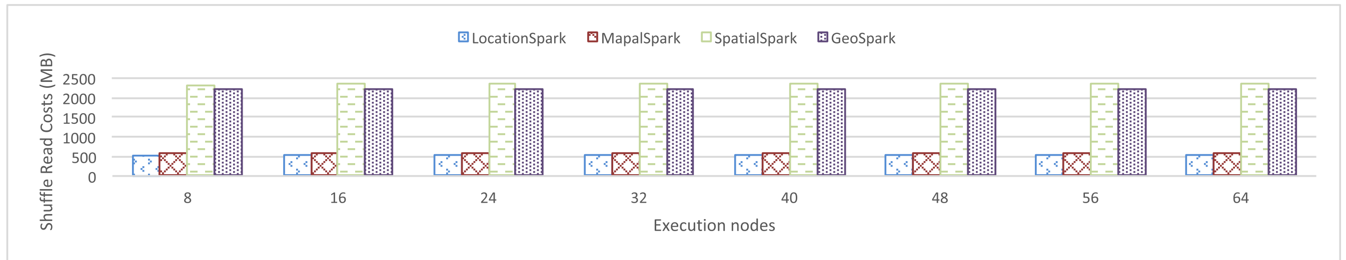
All the experiments were conducted on a Big Data cluster consisting of 16 nodes. Each node has the following features: 2 x CPU 2.2GHz, 384 GB RAM 2400MT/s and 32 TB HDD 6Gbps. Spark applications were deployed in cluster mode. The resource manager was YARN. Two different versions of Spark, 1.6 and 2.2, were used.

TABLE 1. Spatial resolution and number of points of *raster* datasets.

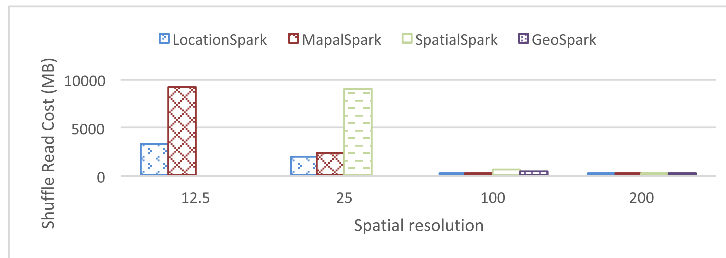
Spatial Resolution (meters)	Size (# cells)
200	1165824
100	4661184
50	18653375
25	74622330
12.5	298472418

For these experiments, elevation data at different spatial resolutions are used as input *raster* data, whereas a combustion model dataset is used as input vector *polygon* data. Spatial resolutions (in meters) of *raster* datasets and relevant size (number of points) are shown in Table 1. Polygon dataset contains 11057 *polygons*, with an average of 175 points in the boundary of each polygon. Both elevation dataset and Combustion Model dataset are publicly accessible in the databases of the National Geographic Institute of Spain (https://www.ign.es/web/ign/portal/inicio).

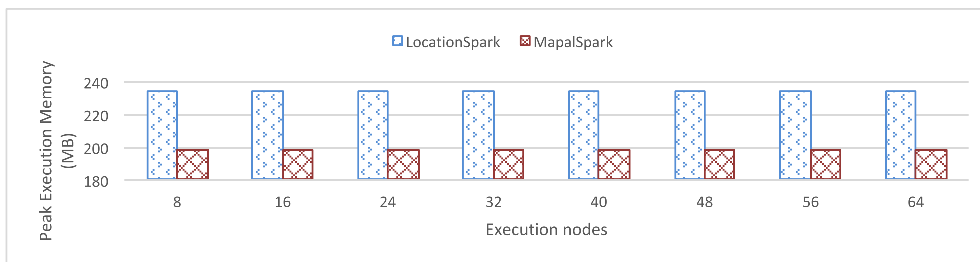
To ensure a fair comparison between the different distributed spatial data processing frameworks, the input dataset has been pre-processed. Since some existing frameworks do not support columns of non-spatial data types, additional input columns storing non spatial values (e.g., elevation values column) have not been included in the analysis tasks for the remainder solutions, only for the solution proposed in the present paper. Furthermore, since LocationSpark only enables spatial processing of rectangular polygons, each input polygon has been replaced by its *Minimum Bounding*



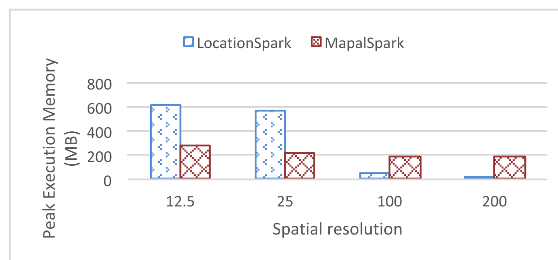
(a) Scalability experiment (spatial resolution of 50 meters)



(b) Data load experiment (40 executors)



(c) Scalability experiment (spatial resolution of 50 meters)



(d) Data load experiment (40 executors)

FIGURE 14. Shuffle read cost and peak execution memory consumption.

Rectangle (MBR). Fig 11 shows the MBR dataset used in this experiment. Additionally, since the rest of existing solutions do not provide integrated *raster-polygon* data analysis, location points within the elevation *raster* are translated to a set of 2D points for each tested solution.

To test the performance of each solution for different workloads, the Spatial Join operation has been executed between the combustion model MBRs and the *raster* datasets shown in Table 1. All frameworks were tested with the following Spark configuration:

- *master*: yarn
- *deploy-mode*: cluster
- *driver-memory*: 8G
- *executor-memory*: 8G

B. PERFORMANCE COMPARISON

The most relevant distributed spatial processing systems in the state of art developed on top of Spark (i.e., GeoSpark [10], LocationSpark [13], SpatialSpark [11] and Stark [16]) have been selected to be compared against the proposed solution.

Fig. 12 shows the execution times of each solution for the spatial join between the polygon dataset and the elevation raster at different spatial resolutions (12m - 400m). A 40-executors configuration has been used

in this experiment. GeoSpark and Stark, Fig. 12 (a), showed the slowest performance. For a proper chart representation, the execution time of Stark for a spatial resolution of 12 meters (7484 s) has not been depicted. LocationSpark and SpatialSpark, Fig. 12 (b), showed a similar behavior. SpatialSpark suffered a Java Heap Memory Overflow at a spatial resolution of 12 meters. Both SpatialSpark and LocationSpark performed more that 4 times faster than Stark for spatial resolutions below 50 meters. The proposed solution, named MapalSpark in the figures, showed the fastest performance, Fig. 12 (c). For a spatial resolution of 12 meters, MapalSpark performed more than one order of magnitude faster than LocationSpark, and more than 2 orders of magnitude faster than Stark.

In addition to the test for different data loads, a scalability test was also performed. For this test a spatial resolution of 50 meters was selected. Scalability behavior of tested solutions for cluster configurations with different number of nodes are plotted in Fig. 13. Although showing the slowest behavior again Stark, Fig. 13 (a), has a good scalability performance from 2 to 16 executors. Then, execution times remain constant. GeoSpark showed similar execution times to Stark but its scalability behavior is much worse. Again, LocationSpark and SpatialSpark, Fig. 13 (b), showed similar execution times. Regarding scalability behavior, SpatialSpark is better than LocationSpark from 2 to 16 executors. Then SpatialSpark remains almost constant but LocationSpark keeps improving until 40 executors. MapalSpark execution times are the fastest again, Fig. 13 (c). Scalability behavior of MapalSpark is very good from 2 to 48 executors. Then, it remains also constant.

Additional performance parameters have been studied. *Shuffle Read Cost* provides information about the amount of data read by executors at the beginning of a Spark stage. Fig. 14 (a) depicts the shuffle read costs of tested solutions for the scalability experiment. Notice that we found that shuffle read costs are equal to shuffle write costs for all solutions, contrary to what is shown in [25], where shuffle read costs showed to be bigger than shuffle read costs for SpatialSpark. As expected, shuffle read cost remained constant regardless the number of executors because the data load also remained constant (spatial resolution of 50 meters has been fixed). The best solution is Stark, it showed no shuffle costs at all. LocationSpark and MapalSpark showed a similar behavior with a cost about 500MB. SpatialSpark and GeoSpark also showed a similar behavior with a cost near to 2500MB. Fig. 14 (b) shows the shuffle read costs for the data load experiment. The best behavior was showed by LocationSpark with less than the half of the cost of MapalSpark for a spatial resolution of 12 meters.

The *Peak Execution Memory* is the maximum memory consumption in any stage during the execution. It was already shown in [25], that the best peak execution memory of the remainder solutions is obtained by LocationSpark, much far better than other solutions. Thus, Fig. 14 (c) shows the comparison between LocationSpark and MapalSpark for the scalability experiment. Peak execution memory remains constant in both solutions. In this case, Mapalspark shows a better behavior. For the data load experiment, Fig. 14 (d), LocationSpark showed a better behavior for spatial resolutions of 200 and 100 meters. As spatial resolution increases, LocationSpark performance decreases whereas MapalSpark shows a better support for high data loads.

VII. CONCLUSION

In this paper it is shown how the use of an integrated data model for vector and raster spatial data enables the implementation of large scale vector-raster spatial analysis systems that outperform the state of the art by more than an order of magnitude. In particular, a naive implementation of the model on top of a combination of Apache Parquet with Apache Spark is described. Next, it is shown how an optimization strategy based on compact range representation for data references and raster points enables achieving the performance results reported above. Performance comparison is done through experiments with different configuration for both the spatial resolution of the raster dataset and the number of nodes of the cluster. Beyond the impressive response times, the proposed solution shows also good behavior in shuffle read and write cost and in peak memory consumption, being competitive in general with the best current solutions. Future work is mainly related to the completion of a prototype system that combines the proposed optimizations with spatial partition and indexing techniques, achieve efficient implementations of other operators such as range queries, joins with different predicates, and kNN queries.

REFERENCES

[1] R. O. Obe and L. S. Hsu, *PostGIS Action*, 2nd ed. Greenwich, CT, USA: Manning Publications Co., 2015.

[2] A. G. Ravi Kothuri and E. Beinat, *Pro Oracle Spatial for Oracle Database 11G*. Berkeley, CA, USA: Apress, 2007.

[3] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, "The multidimensional database system RasDaMan," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, 1998, pp. 575–577, doi: 10.1145/276304.276386.

[4] P. G. Brown, "Overview of sciDB: Large scale array storage, processing and analysis," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2010, pp. 963–968, doi: 10.1145/1807167.1807271.

[5] B. S. A. LaPlante, *Architecting Data Lakes: Data Management Architectures for Advanced Business Use Cases*. Sebastopol, CA, USA: O'Reilly, 2016.

[6] C. Lam, *Hadoop Action*. Stamford, CT, USA: Manning Publications, 2010.

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2Nd USENIX Conf. Hot Topics Cloud Comput.*, Berkeley, CA, USA, 2010, p. 10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[8] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop GIS: A high performance spatial data warehousing system over mapreduce," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1009–1020, Aug. 2013, doi: 10.14778/2536222.2536227.

[9] A. Eldawy and M. F. Mokbel, "SpatialHadoop: A mapreduce framework for spatial data," in *Proc. IEEE 31st Int. Conf. Data Eng.*, Apr. 2015, pp. 1352–1363.

[10] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *Proc. 23rd SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, New York, NY, USA, 2015, pp. 701–704, doi: 10.1145/2820783.2820860.

[11] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in Cloud," in *2015 31st IEEE Int. Conf. Data Eng. Workshops*, Apr. 2015, pp. 34–41.

[12] *Magellan: Geospatial Analytics Using Spark*. Accessed: Nov. 24, 2018. [Online]. Available: https://github.com/harsha2010/magellan
[13] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Location-Spark: A distributed in-memory data management system for big spatial data," *Proc. VLDB Endowment*, vol. 9, no. 13, pp. 1565–1568, Sep. 2016, doi: 10.14778/3007263.3007310.
[14] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 1071–1085, doi: 10.1145/2882903.2915237.
[15] *GeoTrellis: A Geographic Data Processing Engine for High Performance Applications*. Accessed: Nov. 24, 2018. [Online]. Available: https://geotrellis.io/
[16] S. Hagedorn and T. Räth, "Efficient spatio-temporal event processing with STARK," in *Proc. 20th Int. Conf. Extending Database Technol. (EDBT)*, Venice, Italy, Mar. 2017, pp. 570–573, doi: 10.5441/002/edbt.2017.72.
[17] *Apache Parquet*. Accessed: Nov. 24, 2018. [Online]. Available: https://parquet.apache.org/
[18] *Information Technology—Database Languages—SQL Multimedia and Application Packages—Part 3: Spatial*, Standard ISO/IEC 13249-3:2011, International Organization for Standardization (ISO), 2011.
[19] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "MonetDB: Two decades of research in column-oriented database," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 40–45, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/debu/debu35.html#IdreosGNMMK12
[20] *Mongo DB*. Accessed: Nov. 24, 2018. [Online]. Available: https://www.mongodb.com/
[21] M. Ben Brahim, W. Drira, F. Filali, and N. Hamdi, "Spatial data extension for Cassandra NoSQL database," *J. Big Data*, vol. 3, no. 1, p. 11, Jun. 2016, doi: 10.1186/s40537-016-0045-4.
[22] Y. Zhang, M. Kersten, and S. Manegold, "SciQL: Array data processing inside an RDBMS," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, 2013, pp. 1049–1052, doi: 10.1145/2463676.2463684.
[23] S. Villarroya, J. R. R. Viqueira, M. A. Regueiro, J. A. Taboada, and J. M. Cotos, "SODA: A framework for spatial observation data analysis," *Distrib. Parallel Databases*, vol. 34, no. 1, pp. 65–99, Mar. 2016, doi: 10.1007/s10619-014-7165-7.
[24] L. Alarabi and M. F. Mokbel, "A demonstration of ST-Hadoop: A MapReduce framework for big spatio-temporal data," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1961–1964, Aug. 2017, doi: 10.14778/3137765.3137819.
[25] V. Pandey, A. Kipf, T. Neumann, and A. Kemper, "How good are modern spatial analytics systems?" *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1661–1673, Jul. 2018, doi: 10.14778/3236187.3236213.
[26] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in spark," in *Proc. 2015 ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2015, pp. 1383–1394, doi: 10.1145/2723372.2742797.
[27] P. M. D. Gray, *The Functional Approach to Data Management: Modeling, Analyzing, and Integrating Heterogeneous Data*. New York, NY, USA: Springer-Verlag, 2004.
[28] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden, "Materialization strategies in a column-oriented DBMS," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, Apr. 2007, pp. 466–475.

SEBASTIÁN VILLARROYA received the B.S. and M.S. degrees in telecommunication engineering from the Universidade de Vigo, in 2007, and the Ph.D. degree in information technologies from the Universidade de Santiago de Compostela (USC), in 2018. In 2019, he was a Research Associate with Jacobs University Bremen, working on the integration of machine learning algorithms and array databases. Since 2020, he has been a Research Associate with the Centro Singular de Investigación en Tecnoloxías Intelixentes (CITIUS), USC. Beyond distributed big data analysis, sensor data acquisition systems, and big spatial data analytics, he is focused on the integration of machine learning technologies and raster database management systems.

JOSÉ R. R. VIQUEIRA received the master's and Ph.D. degrees in computer science from the University of A Coruña, in 1998 and 2003, respectively. He is currently an Associate Professor with the Department of Electronics and Computer Science, Universidade de Santiago de Compostela (USC), where he is also a Founding Member of the Computer Graphics and Data Engineering (COGRADE) Research Group. He is also a member of the Research Staff of the Centro Singular de Investigación en Tecnoloxías Intelixentes (CITIUS). During his career, he has been a member of three other different research groups of three different universities, namely the Informatics Laboratory, Agricultural University of Athens (Chorochronos Research Project), the Databases Laboratory, University of A Coruña, and the Systems Laboratory, USC. He is the author of numerous publications on different topics related to spatial and spatio-temporal data management applied to GIS. He is also a Founding Partner of Enxenio S.L., a spin-off of the University of A Coruña. His current research interests include the management of very large scientific datasets, with special emphasis on spatio-temporal and environmental data.

JOSÉ M. COTOS has been a Professor with the Department of Electronics and Computing, Universidade de Santiago de Compostela, since 1993, where he is currently a Research Staff with the Centro Singular de Investigación en Tecnoloxías Intelixentes (CITIUS) (http://citius.usc.es). He is also the Coordinator of the Computer Graphics and Data Engineering Research Group. He has participated in more than 20 research projects and in more than 50 contracts with companies and institutions, mostly related to the transfer of technology to the business sector. From 2009 to 2013, he was attached to the presidency of a university network for technology transfer, RedEmprendia. In addition, he was the Founding Partner and an Administrator of the spin-off Paralaxe, Multimedia and Virtual Systems S.L., a spin-off company of the Institute of Technological Research, University of Santiago de Compostela, that was dedicated to the development of multimedia and virtual reality computer systems. He is also involved in the machine learning implementation to industrial processes.

JOSÉ A. TABOADA received the Graduate degree in electronics from the Faculty of Physics, in 1990, and the Ph.D. degree in applied physics from the Universidade de Santiago de Compostela (USC), Spain, in 1996. He is currently an Associate Professor with the Department of Electronics and Computing, USC, where he is also a member of the Computer Graphics and Data Engineering (COGRADE) Research Group. He is also a member of the Centro Singular de Investigación en Tecnoloxías Intelixentes (CITIUS), USC. He is the author of numerous publications and he has been a part of several national and European projects in the fields of intelligent systems. His research interest includes big data.