Fields as a Generic Data Type for Big Spatial Data

Gilberto Camara[1,2], Max J. Egenhofer[3], Karine Ferreira[1], Pedro Andrade[1], Gilberto Queiroz[1], Alber Sanchez[2], Jim Jones[2], Lúbia Vinhas[1]

[1] Image Processing Division, National Institute for Space Research (INPE), Av. dos Astronautas, 1758, 12227-001 São José dos Campos, Brazil
[2] Institute for Geoinformatics (ifgi), University of Münster, Heisenbergstraße 2, 48149 Münster, Germany
[3] National Center for Geographic Information and Analysis and School of Computing and Information Science, University of Maine, Orono, ME 04469-5711, USA

Abstract. This paper defines the Field data type for big spatial data. Most big spatial data sets provide information about properties of reality in continuous way, which leads to their representation as fields. We develop a generic data type for fields that can represent different types of spatiotemporal data, such as trajectories, time series, remote sensing and, climate data. To assess its power of generality, we show how to represent existing algebras for spatial data with the Fields data type. The paper also argues that array databases are the best support for processing big spatial data and shows how to use the Fields data type with array databases.

Keywords: field data type, spatial fields, spatiotemporal data, big spatial data

1 Introduction

One of the biggest changes in Geoinformatics in recent years arises from technologies that produce lots of data. Earth observation and navigation satellites, mobile devices, social networks, and smart sensors create large data sets with space and time references. Big spatial data enables real-time applications, such as tracking environmental changes, detecting health hazards, analyzing traffic, and managing emergencies. Big data sets allow researchers to ask new scientific questions, which is both an opportunity and a challenge [2]. However, there are currently no appropriate conceptual models for big spatial data. Lacking sound guidance, we risk building improvised and incompatible application, with much effort wasted.

A model for big spatial data should consider the nature of the data, which are records of measurements and events in space-time. Sensors measure properties of nature, such as temperature, soil moisture, and land surface reflectance, and human events, such as locations of people and cars. Since these sensors observe

the world in real-time, we take big spatial data to be records of continuous phenomena.

The terms *fields* and *coverages* describe real-world phenomena that vary continuously in space and time [5,9,26]. Despite the abstract nature of the concept, most work on fields deals with concrete data structures (e.g., triangulations, cells, and contours). The OGC definition for coverages–"digital spatial information representing space-time varying phenomena" [25]–is similar to the definition of the Fields data type. Since OGC's coverages focus on describing operations on concrete spatial representations, they add complexity and reduce generality [24,25]. Big spatial data, however, needs an inclusive model that starts with the measurements (i.e., the data collected) and builds on top of them a generic scheme for space-time analyses. The lack of such a high-level model is a serious impediment in analyses of large, complex, and diverse data sets. To avoid makeshift approaches, one needs a wide-ranging, yet simple model for fields at a higher abstraction level than implementation-specific solutions.

Early work on spatial data modeling viewed *fields* as four-dimensional functions $f(x, y, z, t)$ that describe positions in space-time [14,20]. This approach was later refined with *geo-atoms*, the minimal form common to all geographic information and a basis for modeling spatial phenomena [15]. A geo-atom combines a position in space–time and a property, expressed as a tuple $[\mathbf{x}, \mathbf{Z}, z(\mathbf{x})]$, where $\mathbf{x}$ is a position in space–time, $\mathbf{Z}$ is a property, and $z(\mathbf{x})$ is the value of the property at that position. To represent fields, we take the idea of geo-atoms one step further and consider how one observes reality. Since one will never have complete information about external reality, one needs to make inferences about positions in space-time for which there are no observations [21]. Thus, field representations have to combine *observed* and *inferred* measures of a phenomenon. One needs to put together observations of properties of reality with a procedure that estimates values of these properties at non-observed positions [8].

This paper defines *fields* as *sets of geo-atoms* $\{[\mathbf{x}, \mathbf{Z}, z(\mathbf{x})]\}$ that are observations of a property Z in an space-time extent, and an *estimator function* that estimates values of this property in non-observed locations of this extent. A field has a *space-time* extent, a set of *positions* inside this extent, and a set of *values* observed or estimated for each position. We define a Field data type based on abstract specifications, following a line of research in Geoinformatics that considers formal definitions precede reliable system implementation [12,11,32].

Although the Field data type is not specific for dealing with big spatial data, it is particularly relevant for handling large data sets. Contemporary object-relational data models are built around layers, which slice the geographic reality in a particular area. The use of layers as a basis for spatial data organization comes from how data is organized in thematic and topographic maps. When applied to big spatial data, the organizing principle of geographic layers breaks down, however. Instead of a set of static spatial layers (each with its legend), big spatiotemporal data sets store information about changes in space and time. Conceiving such information as fields captures their inherent nature better than the traditional layer-oriented view.

After a brief discussion on generic programming and generic types in Section 2, we introduce the *Field data type* (Section 3). We show how to use the Field data type to represent time series, sensor networks, trajectories, collections of satellite images, and climate data, sharing common operations. Section 4 shows how to implement existing spatiotemporal algebras using the Field data type. Section 5 discusses the nature of big spatial data; we make a case for array databases as the best current support for handling these data sets. Section 6 shows how to use the Field data type in connection with array databases for processing large spatial data. The paper closes with a discussion of a road map for making the Field data type a tool for developing new types of GIS applications.

2 Generic Programming and Generic Types

The design of the Field data type is based on the ideas of generic programming. In this style of software development, one first specifies the required algorithms, and then makes them generic so they work for different types and data structures [23,30]. This style is well-suited for building a GIS. Most spatial algorithms do not depend on any particular spatial data structure, but only on a few fundamental semantic properties of that structure. Such properties include the ability to get from one item of the data structure to the next, and to compare two items. To find the mean value of an attribute in a spatial data set, it is irrelevant whether the data structure is a TIN, a grid, or a set of polygons. All one needs is to look into a list of values and compare them. Even algorithms that depend on spatial properties can be expressed in an abstract form. One can define the local mean of a data set using an abstract definition of neighborhood, leaving the details to the implementation phase. Thus, GIS software design can gain a lot from generic programming.

Generic programming uses *abstract data types*, which are formal tools that allow an objective evaluation of computer representations [3]. Type definitions have an externally viewable set of operations and a set of axioms applicable to them [16]. For this purpose, we use the following notation. Type definitions and operations use a `monospaced` font. Type names are capitalized (e.g., `Integer`). Sets of instances of a type are included in braces, for instance, `{Integer}` is a set of variables of type `Integer`. We write an ordered pair of variables of types A and B as (`TypeA`, `TypeB`).

Generic types are indicated by `T:GenericType`, where `T` is a placeholder for a concrete type. The notation `I:Item` defines a generic type of items, where the concrete type can be, for example, `Integer` or `Real`. Types that use other generic types are written as `CompositeType [T:GenericType]`, so `Stack[I:Item]` defines a composite type `Stack` that handles instances of the generic type `I:Item`.

To associate concrete types to a generic type, we write `T:GenericType` $\models$ `ConcreteTypeA, ConcreteTypeB`. To point out that one can replace the generic type `I:Item` by concrete types `Integer` and `Real`, we write `I:Item` $\models$ `Integer, Real`.

Names of functions and operators begin with a lowercase letter. Examples are `top`, `pop`, and `new`. Function signatures point out their input types and the output type. The notation `(TypeA x TypeB → TypeC)` describes a function where `TypeA` and `TypeB` are the types of the input and `TypeC` is the type of the output. A `factorial` function has (`Integer → Integer`) as a signature. Functions can use generic types. A generic `sum` function has `I:Item x I:Item → I:Item` as a signature.

Consider a stack, a last-in, first-out data structure. It has three fundamental operations: `push`, `pop`, and `top`. The `push` operation adds an item to the top of the stack, `pop` removes the item from the top of the stack, while `top` returns the element at the top of the stack, without changing the stack. The stack specification (Fig. 1) provides a consistent and concise description of its behavior. The basic operations `push`, `pop`, and `top`, and the auxiliary functions `new` and `isEmpty`, are defined independently of the data structures and algorithms that implement them. This specification provides support for implementing stacks of different concrete types (e.g., stacks of integers, stack of strings, or stacks of any other user-defined type including stacks of stacks).

```
Type Stack [I] uses I:Item
Functions
    new: Stack
    push: I x Stack → Stack[I]
    pop: Stack[I] → Stack[I]
    isEmpty: Stack[I] → Boolean
    top: Stack[I] → i
Variables
    s: Stack
    i: Item
Axioms
    isEmpty (new ()) = true
    isEmpty (push(i, s)) = false
    top (new ()) = error
    top (push (i, s)) = i
    pop (push (i, s)) = s
```

Fig. 1: Abstract specification of the data type stack

3 Fields as Generic Types

What is in common between a time series of rainfall in Münster, the trajectory of a car in Highway 61, a satellite image of the Amazon, and a model of the Earth's climate? They share the same inherent structure. They all have a space-time extent, within which one measures values of a phenomenon, providing observations of reality. Within this extent, one can also compute the values of these

phenomena at non-observed positions. We thus conceptualize these data sets as *fields*, made of *sets of geo-atoms* $\{[\mathbf{x}, \mathbf{Z}, z(\mathbf{x})]\}$ that are observations of a property Z in an space-time extent, and an *estimator function* that estimates values of this property in non-observed locations of this extent.

This definition of fields is a generalization of the traditional view of fields as functions that map elements of a bounded set of locations in space onto a value-set [13]. We extend this idea in two ways: (1) we consider different types of locations in space and time and (2) we consider that the elements of the value-set can also be positions in space-time. Thus, a field is a function whose domain is an extent of space-time, where one can measure the values of a property in all positions inside the extent.

The key step in this conceptualization is the generic definition of the concepts of *position* and *value*. In a time series of rainfall, positions are time instants, since space is fixed (the sensor's location), while values are the precipitation counts. In a remote sensing image, positions are samples in 2D space (the extent of the image), since time is fixed (the moment of image acquisition), while values are attributes, such as surface reflectance. Logistic and trajectory models record moving objects by taking positions as time instances, while their values are the objects' locations in space.

Type definitions need the following building blocks:

```
P:Position ⊨  Instant, 2DPoint, 3DPoint,
              (2DPoint, Instant), (3DPoint, Instant)
V:Value    ⊨  Integer, Real, Boolean, String, P:Position
E:Extent   ⊨  [(3DCube, Interval)], [(3DPolygon, Interval)]
```

Fig. 2: Building blocks for the Fields data type

The generic type `P:Position` stands for positions in space-time. This type is mapped onto concrete types that express different time and space cases. Some non-exhaustive examples are `Instant` for time instants, `2DPoint` and `3DPoint` for purely spatial positions, and pairs `(2DPoint, Instant)` and `(3DPoint, Instant)` for space-time positions. The generic type `V:Value` stands for attribute values. Concrete types linked to `V:Value` include Integer, Real, String, Boolean and their combination. Values can also be associated to positions, as in the case of trajectories.

Each field exists inside an extent of space-time, represented by the type `E:Extent`, whose instances are sets of 3D compact regions in space-time. Each field has an associated `G:Estimator` function that enables estimating values at positions inside its extent. This allows a field to infer measures at all positions inside the extent. The estimator function use the field's information and thus has a signature `(F:Field x P:Position` $\rightarrow$ `V:Value)`.

The relationship between positions and extents is a key part of the model. All positions of a field are contained inside the extent. Thus, the possible concrete

types for the generic type Position are those that can be topologically evaluated as being part of a space-time hypercube or a space-time polygon. The definition of an extent as a set of space-time hypercubes also avoids the problems with null values. Thus, there are no null values inside a field extent in this Field model.

The Field definition is independent of granularity, which we take to be a problem-dependent issue. Each concrete field will have its spatial and temporal granularity that will determine how its operations are implemented. Temporal granularity will be represented by the concrete implementation of types `Interval` and `Instant`. The granularity of type Instant should be such that it is always possible to test whether an instant is inside an interval.

The formal description of a Field data type is shown in Fig. 3. The operations of the Field data type are:

new Creates a new Field, given an extent and an estimator function.
add Adds one observation with a (position, value) pair to the Field.
obs Returns all observations associated to the Field.
domain Returns the full set of positions inside the Field's extent. The actual result of this operation depends on the Field's granularity, but the operation can be defined in a problem-independent way.
extent Returns the extent of the Field.
value Computes the value of a given position, using the `estimator` function. The estimator ensures that a field will represent a continuous property inside its extent.
subfield Returns a subset of the original Field according to an extent. This function is useful to retrieve part of a Field.
filter Returns a subset of the original Field that satisfies a restriction based on its values. Examples include functions such as 'values greater than the average.'
map Returns a new Field according to a function that maps values from the original Field to the field to be created. Examples of `map` include unary functions such as `double` and `squareRoot`. This function corresponds to a `map` in functional programming.
combine Creates a new Field combining two fields with the same extent, according to an operation to be applied for each element of the original Fields. Examples of `combine` include binary functions such as `sum` and `difference`.
reduce Returns a value that is a combination of all the values of some positions the Field. Examples include statistical summary functions such as `maximum`, `minimum`, and `mean`.
neigh Returns the neighborhood of a position inside a Field. It uses a function that compares two positions and finds out whether they are neighbors. One example of the function is a proximity matrix where each position is associated to all its neighbors.

The Fields data type distinguishes between the extent and the domain of a field. The extent is the region of space-time where one is able to get a value for each position. The domain of a field is the set of positions it contains, whose granularity depends on how the field was constructed. For example, two fields

may have the same extent and different domains. For the same extent, one field may have a set of scattered position as its domain, while another may have ist positions organized in a regular grid in space-time. One can perform operations between these fields without changing their granularities, since they adhere to the same operations.

```
Field [E, P, V, G] uses E:Extent, P:Position, V:Value, G:Estimator
 Operations
 new:          E x G → Field
 add:          Field x (P, V) → Field
 obs:          Field → {(P, V)}
 domain:       Field → {P}
 extent:       Field → E
 value:        Field x P → V
 subfield:     Field x E → Field
 filter:       Field x (V → Bool) → Field
 map:          Field x (V → V) → Field
 combine:      Field x Field x (V x V → V) → Field
 reduce:       Field x (V x V → V) → V
 neigh:        Field x P x (P x P → Bool) → Field
 Variables
 f, f1, f2: Field
 g: Estimator
 p: Position
 e: Extent
 v: Value
 Functions
 uf: (V → V)         -- unary function on values
 bf: (V x V → V)     -- binary function on values
 ff: (V → Bool)       -- filter function on values
 nf: (P x P → Bool) -- neighborhood function on positions
 Axioms
 -- basic fields axioms: a field is dense relative to its extent
 ∀ p ∈ extent(f) ⟹ ∃ value(f, p) = g(f, p)
 ∀ p ∉ extent(f) ⟹ value(f, p) = ∅
 -- axioms on operation behavior
 ∀ f, domain(f) ⊆ extent (f)
 subfield(f, e) ⊆ f ⟺ e ⊆ extent (f)
 filter(f, ff) ⊆ f
 obs (new(e, g)) = ∅
 obs (add (new(e, g)), (p, v))) =
    (p, v) ⟺ p ⊂ e
 subfield(f, extent(f)) = f
 neigh (f, p, nf) ⊆ f, ∀ p ∈ extent (f)
 value (map (f, uf), p) =
    uf (value (f, p)), ∀ p ∈ extent (f)
 value (combine(f1, f2, bf), p) =
    bf (value (f1, p), value (f2, p)) ⟺
    p ∈ extent (f1) and p ∈ extent (f2)
 reduce (f, bf) =
    bf (reduce (f1, bf), reduce(f2, bf)) ⟺
    f1 = subfield (f,e1) and f2 = subfield(f,e2) and
    e1 ∩ e2 = ∅ and e1 ∪ e2 = extent (f)
```

Fig. 3: Generic data type definition of Field

4 Implementing Existing Algebras with the Fields Data Type

To show how to use the Fields data type, we consider how to express two existing algebras for spatial data using it: Tomlin's map algebra [31] and the STAlgebra [8]. Map Algebra is a set of procedures for handling continuous spatial distributions. It has been generalized to temporal and multidimensional settings [10,4,22]. Tomlin defines the following map algebra operations:

Local functions: The value of a location in the output map is computed from the values of the same location in one or more input maps.

Focal functions: The value of a location in the output map is computed from the values of the neighborhood of the same location in the input map.

Zonal functions: The value of a location in the output map is computed from the values of a spatial neighborhood of the same location in an input map. This neighborhood is a restriction on a second input map.

Fig. 4 shows how to express Tomlin's map algebra functions with the Field data type.

```
Variables
f1, f2: Field                          -- input
f3: Field                              -- output
p, p1: Position
Functions
uf: (v:Value → v:Value)                -- unary function
bf: (v:Value x v:Value → v:Value)      -- binary function
nf: (p:Position x p:Position → Bool) -- neighborhood function
Operators
localUnary (f1, uf) = map (f1, uf)
localBinary (f1, f2, bf) = combine (f1, f2, bf)
focalFunction (f1, nf, bf) =
   ∀ p ∈ domain (f3)
     add(f3, (p, reduce (neigh (f1, p, nf), bf)))
zonalFunction (f1, f2, nf, bf) =
   ∀ p ∈ domain(f3)
     add(f3, (p, reduce (subfield (f1,
                            extent (neigh (f2, p, nf), bf)))))
```

Fig. 4: A generic map algebra

To implement a generic map algebra, the *local unary* and *local binary* functions are mapped onto the `map` and `combine` operators, respectively. Local functions involving three or more maps can be broken down into unary and binary functions. A *focal function* uses the functions `neigh` and `reduce`. The `neigh` function returns a field with only those local values that are used by `reduce` to get a

new value for the position in the output field. The same combination implements *zonal functions*. The difference is that the neighborhood function is defined on a second field. The extent of the neighborhood of the second field is used to extract a subfield of the first field. The function `reduce` then produces a unique value that is the new value of the position in the output field. The mapping is dimension-independent and can be used to implement not only Tomlin's 2D map algebra [31], but also a multidimensional map algebra [22] and a temporal map algebra [10].

A second example is STAlgebra [8], which takes observations as its basic building blocks. Based on Sinton's view of the inherent nature of geographical data [28], STAlgebra singles out different types for spatiotemporal data: `Coverage`, `CoverageSeries`, `TimeSeries`, and `Trajectories`. Operations on these types allow queries and inferences on space-time data. Instances of these types can be related to *events*. The mappings from the four spatiotemporal data types `TimeSeries`, `Trajectory`, `Coverage` and `CoverageSeries` onto the `Field` type are as follows:

Time Series A *time series* represents the variation of a property over time in a fixed location. For example, a time series of rainfall has measured values of precipitation counts at some controlled times (e.g., hourly) at the sensors' locations. A `TimeSeries` type is mapped onto a `Field[E:Extent, Instant, V:Value, G:Estimator]` where positions are time instants.

Trajectory A *trajectory* represents how locations or boundaries of an object evolve over time. For example, a trajectory of an animal, which has a fixed identification, is composed of measured spatial locations at controlled times (e.g., hourly). The `Trajectory` type of STAlgebra is mapped onto `Field[(3DPolygon, Interval), Instant, 2DPoint, G:Estimator]` or onto a `Field[ ( 3DPolygon, Interval), Instant, 3DPoint, G:Estimator]`, if the trajectory is taken in 2D or 3D space, respectively.

Coverage A *coverage* represents the variation of a property within a spatial extent at a certain time. A remote sensing satellite image is an example of a coverage. It has a fixed time, the moment of the image acquisition, and measured values of surface reflectance at spatial locations. The `Coverage` type is mapped onto a `Field[E:Extent, 2DPoint, V:Value, G:Estimator]` whose positions are 2D spatial locations.

CoverageSeries A *coverageseries* represents a time-ordered set of coverages that have the same boundary. A sequence of remote sensing satellite images over the same region is an example of a coverage series. The `CoverageSeries` type has a fixed spatial extent and measured coverages at controlled times. It is mapped onto a `Field[E:Extent, (2DPoint, Instant), V:Value, G:Estimator]` whose positions have variable 2D spatial locations and times. The field's extent is composed of the coverage series' spatial extent and an interval that encloses all position instances.

5 Array Databases for Big Spatial Data

Big spatial data comes from many different sources and with different formats. Among those sources are Earth Observation satellites, GPS-enabled mobile devices and social media. For example, the LANDSAT data archive at the United States Geological Survey has more than 5 million images of data of the Earth's land surface, collected over 40 years, comprising about 1 PB of data. These data sets allow researchers to explore big data sets for innovative applications. One example is the world's first forest cover change map from 2000 to 2012 at a spatial resolution of 30 meters [17].

The challenge for handling big spatial data is to design a programming model that can be scaled up to petabyte data sets. Currently, most scientific data analysis methods for Earth observation data are file-based. Earth observation data providers offer data to their users as individual files. Scientific and application users download scenes one by one. For large-scale analyses, users need to obtain hundreds or even thousands of files. To analyze such large data sets, a program has to open each file, extract the relevant data and then move to the next file. The program can only begin its analysis when all the relevant data has been gathered in memory or in intermediate files. Data analysis on large datasets organized as individual files will run slower and slower as data volumes increase. This practice has put severe limits on the scientific uses of Earth Observation data.

To overcome these limitations, there is a need for a new type of information system that manages large Earth Observation data sets in an efficient way and allows remote access for data analysis and exploration. It should also allows existing spatial (image processing) and temporal (time series analysis) methods to be applied to large data sets, as well as enabling development and testing of new methods for space-time analyses of big data. After analyzing alternatives, such as MapReduce [7], we consider that array databases offer the best current solution for big spatial data handling. Array databases offer a model of programming that suits many of tasks for analysis of spatiotemporal data.

Array databases organize data as a collection of arrays, instead of tables used in object-relational DBMSs. Arrays are multidimensional and uniform, as each array cell holds the same user-defined number of attributes. Attributes can be of any primitive data type such as integers, floats, strings or date and time types. To achieve scalability, array databases strive for efficiency of data retrieval of individual cells. Examples of array databases include RasDaMan [1] and SciDB [29].

Array databases have no semantics, making no distinction between spatial and temporal indexes. Thus, to be used in spatial applications, one needs to extend them with types and operations that are specific for spatiotemporal data. That is where the Fields data type is particularly useful.

6 Fields Operations in Array Databases

This section shows how to map the fields data type onto the array database SciDB [29]. SciDB splits big arrays into chunks that are distributed among different servers; each server controls a local data storage. One of the instances in the cluster is the coordinator, responsible for mediating client communications and for orchestrating query executions. The other instances, called workers, participate in query processing. SciDB takes advantage of the underlying array data model to provide an efficient storage mechanism based on chunks and vertical partitions. Compared to object-relational databases, the SciDB solution provides significant performance gains. Benchmarks comparing object-relational databases and array databases for big scientific data have shown gains in performance of up to three orders of magnitude in favor of SciDB [6,27].

SciDB provides two query languages: an Array Query Language (AQL) that resembles SQL and an Array Functional Language (AFL) closely related to functional programming. There are two categories of functions:

scalar functions Algebraic, comparison and temporal functions, that operate over scalar values.

aggregates Functions that operate on array level, like average, standard deviation, maximum and minimum values.

Natively, SciDB already supports some of the operations of the Fields data type. The operations of the Fields data type currently available in SciDB are described in Table 1. We tested these operations using arrays of different sizes, as discussed below.

Field op	signature	SciDB op
`map`	`Field x (v:Value → v:Value)`	`apply`
`subfield`	`Field x e:Extent → Field`	`subarray`
`filter`	`Field x (v:Value → Bool)`	`filter`
`reduce`	`Field x ({v:Value} → v:Value)`	`aggregate`

Table 1. Fields model mapped onto SciDB

Our evaluation used a set of images from the MODIS sensor, which flies onboard NASA's Terra and Aqua remote sensing satellites. The MODIS instruments capture data in 36 spectral bands. Together the instruments image the entire Earth every 1 to 2 days. They are designed to provide measurements in large-scale global dynamics, including changes in the Earth's cloud cover, radiation budget, and processes occurring in the oceans, on land, and in the lower atmosphere [19].

We used the MODIS09 land product with three spectral bands (visible, near infrared, and quality). Each MODIS09 image is available for download at the NASA website as a tile covering 4,800 x 4,800 pixels in the Earth's surface at 250

meters x 250 meters ground resolution. We then combined twenty time steps of the 22 MODIS images that cover Brazil, giving a total of 440 images that were merged into an array of 10,137,600,000 cells. Each cell contains three values, one for each band. This array was then loaded into SciDB for our experiment.

We first used the SciDB `subarray` function to select subsets of the large array for evaluation purposes. For each subarray, we used the SciDB `apply` function to calculate the enhanced vegetation index [18] associated to each cell and stored the results in a new subarray. Next, we used the `filter` operation to select from each resulting subarray those cells whose red value was greater than 100 and stored the results. Finally, we used the `aggregate` function to calculate the average of the one attribute of each subarray and store the results. Fig. 5 shows the test results as the average of 10 runs for the following number of cells: $20 * 1024^2, 20 * 2048^2, 20 * 3072^2, 20 * 4096^2, 20 * 5120^2, 20 * 6144^2, 20 * 7168^2, 20 * 8192^2, 20 * 9216^2, 20 * 10240^2, 20 * 11264^2, 20 * 12288^2, 20 * 13312^2, 20 * 14336^2$.

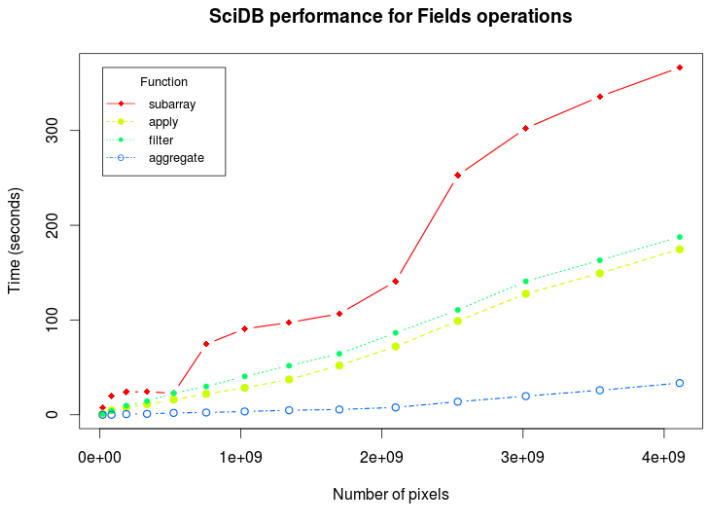

Fig. 5: Performance measures for Field operations in SciDB operations

These results were obtained in a single Ubuntu server, having 1 Intel Xeon 2.00 GHz CPU, with 24 cores and 132 GB memory. The performance results are satisfactory, since the processing time grew roughly linearly with array size. With a bigger server configuration, we can expect better results. These results have given us confidence that combining the Fields data type with array database is viable and likely to produce good results. As part of later work, we will implement the whole Fields data type in SciDB, making it a suitable environment for processing large spatial data.

Although array databases currently offer the most promising approach for handling big spatial data sets, they do not yet offer all of the support required by spatial applications. Most spatial applications need to combine field data sets with information about spatial objects, such as cities and farms. Also, array databases treat all dimensions equally. Therefore, developers of spatial applications need to provide additional support to use array databases effectively. This is a promising new research area that can lead to spatial information infrastructures that will make good use of large data sets.

7 Conclusions

This paper defined the Field abstract data type for representing continuous spatiotemporal data. The motivation was to provide a sound basis for applications that deal with big spatial data sets. These data sets can come for many different sources and have many purposes, yet they share common features: in all of them, one measures values at positions in space-time. The underlying conceptual view is that these data sets are measures of continuous phenomena, thus leading to fields. We showed that the Fields data type can represent data sets, such as maps, remote sensing images, trajectories of moving objects, and time series.

We also considered the problem of how to implement the Field data type operations in an environment suitable for handling large spatial data and argued that array databases are currently the best approach available. Some of the operations of the Field data type are already available in the open source array database SciDB, and our experiments showed that the performance of SciDB is encouraging. Given the results so far, we will implement the full set of the Field data type operations directly in SciDB to provide a full features of Field data type in array databases.

We anticipate that the combination of the Field data type and array databases can bring about a disruptive change in spatial information infrastructures. Consider the case of Earth Observation data. Currently, remote sensing data is retrieved from the data archives on a scene-by-scene basis and most applications use only one temporal instance per geographical reference. In an advanced infrastructure, researchers and institutions will break the image-as-a-snapshot paradigm, as entire collections of image data will be archived as single spatiotemporal arrays. Users will be able to develop algorithms that can span seamless partitions in space, time, and spectral dimensions, and arbitrary combinations of those. These algorithms will provide new insights into changes in the landscape.

We believe that the combination of simple, yet powerful data types with new technologies for spatial data management will bring about large changes in the use of spatial information, especially for data that promotes the public good. Data management of large data sets will be done in petascale centers. Users will have the means to perform analysis and queries on these data sets. Petascale centers that promote open data policies and open data analysis will get large benefits from increased awareness of the value of spatial information for society.

Acknowledgments

Gilberto Camara thanks for the support of the Brazilian research agencies FAPESP (grant 2008/58112-0) and CNPq (grant 04752/2010-0). This work was written while Gilberto Camara was the holder of the Brazil Chair at the University of Münster, supported by the Brazilian agency CAPES (grant 23038.007569/2012-16). Gilberto also received substantial financial and logistical support provided by the Institute of Geoinformatics at the University of Münster, Germany. Max Egenhofer's work was partially supported by NSF Grant IIS-1016740.

References

1. Baumann, P., Dehmel, A., Furtado, P., Ritsch, R., Widmann, N.: Spatio-temporal retrieval with RasDaMan. In: Proceedings of the 25th International Conference on Very Large Data Bases. pp. 746–749. VLDB '99 (1999)
2. Campbell, P.: Editorial on special issue on big data: Community cleverness required. Nature 455(7209), 1 (2008)
3. Cardelli, L., Wegner, P.: On understanding type, data abstraction, and polymorphism. ACM Computing Surveys 17(4), 471–552 (1985)
4. Cordeiro, J., Camara, G., Freitas, U., Almeida, F.: Yet another map algebra. Geoinformatica 13(2), 183–202 (2009)
5. Couclelis, H.: People manipulate objects (but cultivate fields): Beyond the raster-vector debate in GIS. In: Frank, A., Campari, I., Formentini, U. (eds.) Theories and Methods of Spatio-Temporal Reasoning in Geographic Space. LNCS, vol. 639. Springer
6. Cudre-Mauroux, P., Kimura, H., Lim, K.T., Rogers, J., Madden, S., Stonebraker, M., Zdonik, S., Brown, P.: SS-DB: A standard science DBMS benchmark. In: XLDB 2010 - Extremely Large Databases Conference (2012)
7. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Communications ACM 51(1), 107–113 (2008)
8. Ferreira, K., Camara, G., Monteiro, A.: An algebra for spatiotemporal data: From observations to events. Transactions in GIS 18(2), 253–269 (2014)
9. Frank, A.: Tiers of ontology and consistency constraints in geographic information systems. International Journal of Geographical Information Science 15(7), 667–678 (2001)
10. Frank, A.: Map algebra extended with functors for temporal data. In: Akoka, J. (ed.) Perspectives in Conceptual Modeling: ER 2005 Workshop. LNCS, vol. 3770, pp. 194–207. Springer (2005)
11. Frank, A.: GIS theory - the fundamental principles in GIScience: A mathematical approach. In: Harvey, F.J. (ed.) Are there Fundamental Principles in Geographic Information Science? pp. 12–41 (2012)
12. Frank, A., Kuhn, W.: Specifying Open GIS with functional languages. In: Egenhofer, M., Herring, J. (eds.) Advances in Spatial Databases—4th International SSD Symposium. LNCS, vol. 951, pp. 184–195. Springer-Verlag, Berlin (1995)
13. Galton, A.: Fields and objects in space, time and space-time. Spatial cognition and computation 4 (2004)
14. Goodchild, M.: Geographical data modeling. Computers and Geosciences 18(4), 401–408 (1992)

15. Goodchild, M., Yuan, M., Cova, T.: Towards a general theory of geographic representation in GIS. International Journal of Geographical Information Science 21(3), 239 – 260 (2007)
16. Guttag, J., Horowitz, E., Musser, D.: Abstract data types and software validation. Communications of the ACM 21(12), 1048–1064 (1978)
17. Hansen, M., Potapov, P., Moore, Hancher, Turubanova, S., Tyukavina, A., Thau, D., Stehman, S., Goetz, S., Loveland, T., Kommareddy, A., Egorov, A., Chini, L., Justice, C., Townshend, J.: High-resolution global maps of 21st-century forest cover change. Science 342(6160), 850–853 (2013)
18. Jiang, Z., Huete, A., Didan, K., Miura, T.: Development of a two-band enhanced vegetation index without a blue band. Remote Sensing of Environment 112(10), 3833–3845 (2008)
19. Justice, C., Townshend, J., Vermote, E., Masuoka, E., Wolfe, R., Saleous, N., Roy, D., Morisette, J.: An overview of MODIS land data processing and product status. Remote Sensing of Environment 83(1), 3–15 (2002)
20. Kemp, K.: Fields as a framework for integrating GIS and environmental process models. part one: Representing spatial continuity. Transactions in GIS 1(3), 219–234 (1997)
21. Kuhn, W.: Geospatial semantics: Why, of what, and how? Journal of Data Semantics 3, 1–24 (2005)
22. Mennis, J.: Multidimensional map algebra: Design and implementation of a spatiotemporal GIS processing language. Transactions in GIS 14(1), 1–21 (2010)
23. Musser, D., Stepanov, A.: Generic programming. In: Gianni, P. (ed.) Symbolic and Algebraic Computation, Lecture Notes in Computer Science, vol. 358, pp. 13–25 (1989)
24. OGC: The OpenGIS abstract specification - Topic 6: Schema for coverage geometry and functions (Tech. Rep. OGC 07-011). Tech. rep., Open Geospatial Consortium, Inc. (2007)
25. OGC: OGC web coverage service (WCS) interface standard - Core (OGC 09-110r3). Tech. rep., Open Geospatial Consortium, Inc. (2010)
26. Peuquet, D.: Representations of geographic space: Toward a conceptual synthesis. Annals of the Association of American Geographers 78(3), 375–394 (1988)
27. Planthaber, G., Stonebraker, M., Frew, J.: EarthDB: scalable analysis of MODIS data using SciDB. In: Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data. pp. 11–19. ACM (2012)
28. Sinton, D.: The inherent structure of information as a constraint to analysis: Mapped thematic data as a case study. In: Dutton, G. (ed.) Harvard Papers on Geographic Information Systems. vol. 7, pp. 1–17. Addison-Wesley, Reading, MA (1978)
29. Stonebraker, M., Brown, P., Zhang, D., Becla, J.: SciDB: A database management system for applications with complex analytics. Computing in Science & Engineering 15(3), 54–62 (2013)
30. Stroustrup, B.: The C++ Programming Language, 3rd ed. Addison-Wesley Publishing Company (1997)
31. Tomlin, C.: Geographic Information Systems and Cartographic Modeling. Prentice-Hall, Englewood Cliffs, NJ (1990)
32. Winter, S., Nittel, S.: Formal information modelling for standardisation in the spatial domain. International Journal of Geographical Information Science 17(8), 721–741 (2003)