OGRS

rasdaman
raster data manager

Spatio-Temporal Big Data

the rasdaman approach

How to handle and publish multi-dimensional gridded big data using Array DBMS technology and OGC open standards.

Espoo (FI), 12 June 2014

Contributed by

PIERO CAMPALANI
XINGHUA GUO
PETER BAUMANN

Large-Scale Scientific Information Systems (L-SIS)
Jacobs University Bremen (DE)

Credits:

EarthServer

✉ : {*p.campalani,g.xinghua,p.baumann*}@jacobs-university.de
: http://kahlua.eecs.jacobs-university.de/~lsis/

Contents

Acknowledgements

The research leading to the results presented here has received funding from the European Community's Seventh Framework Programme (EU FP7) under grant agreement n. 283610 "European Scalable Earth Science Service Environment (*EarthServer*)".

Chapter 1

Concepts

What is the `rasdaman` approach to Big Data? Often, especially in the fields of remote sensing and geomatics, Big Data is synonym of Big Rasters: huge space-borne/air-borne, multi/hyper-spectral images are literally creating a deluge of bytes. So again, what does `rasdaman` propose to tackle this challenge?

A first key feature is being open and standard. Data access and curation services are moving towards web GIS platforms, from simple visualization and download, to more advanced computations: it is a clear advantage when every service speaks the same language, and this language is usually defined by the Open Geospatial Consortium (OGC, `http://www.opengeospatial.org/`).

`rasdaman` is the reference implementation (as of 2013) of the OGC Web Coverage Service (WCS) [Baumann, 2012] and its team is actively participating in the evolution of the standards within OGC, especially for everything that surrounds the so-called *coverages* [Baumann, 2012].

Aside of that, `rasdaman` is the reference (and only available implementation) of the most exciting extension of the WCS service: the *Processing Extension* [Baumann and Yu, 2014]. This key feature lets you exploit the flexibility of a full query language for coverages to request ad-hoc processing to the server, so that you can minimize bandwidth usage on data transfer and indeed: move the processing to the data [Hey et al., 2009].

Underneath it all, the `rasdaman` Array DBMS ensures ad-hoc optimizations for the access and elaboration of multi-dimensional arrays, being especially prone to the storage of time-series or — more generally — hypercubes of images and gridded datasets.

In the next subsections we will cover all these concepts in more detail: `rasdaman` and its RasQL query language will be described in Section 1.1; the OGC coverage model (GMLCOV) is explained in Section 1.2; finally Section 1.3 will briefly talk about the OGC web services for coverages available with `rasdaman`.

1.1 rasdaman: the RAster DAta MANager

`rasdaman` is a domain-neutral Array Database System: it extends standard relational database systems with the ability to store and retrieve multi-dimensional raster data (arrays) through an SQL-style query language: the `rasdaman` Query Language (RasQL).

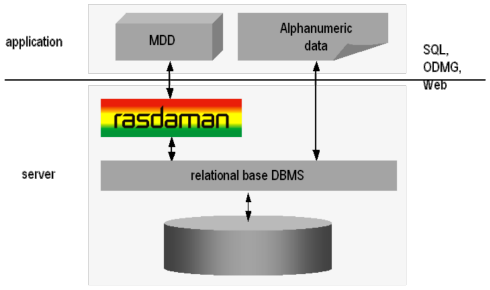

Figure 1.1 – Embedding of `rasdaman` in the IT infrastructure.

Internally and invisible to the application, arrays are decomposed into smaller units by means of customizable tiling strategies, which are then maintained in a conventional DBMS, called the base DBMS. The `rasdaman` open-source project[1] uses PostgreSQL as its base DBMS, and there it stores both the bulk array data and the auxiliary geo-semantics for real-world mapping of the arrays: latitudes, longitudes, time coordinates, resolutions and other ancillary annotations.

But what do we mean as an array?

A multidimensional array (see Figure 1.2) is a set of elements which are ordered in space. The space considered here is discretized, i.e., only integer coordinates are admitted. The number of integers needed to identify a particular position in this space is called the *dimension* (sometimes also referred to as *dimensionality*). Each array element, which is referred to as cell, is positioned in space through its *coordinates*.

A *cell* can contain a single value (such as an intensity value in case of grayscale images) or a composite value (such as integer triples for the red, green, and blue components of a true-color image). All cells share the same structure which is referred to as the *array cell type* or *array base type*.

A *collection* is an heterogeneous bag of arrays, and it does not have to be confused with the gridded *coverage* (see next section) which is stored as a single connected array (easing the maintenance of coherent geometries within the coverage model).

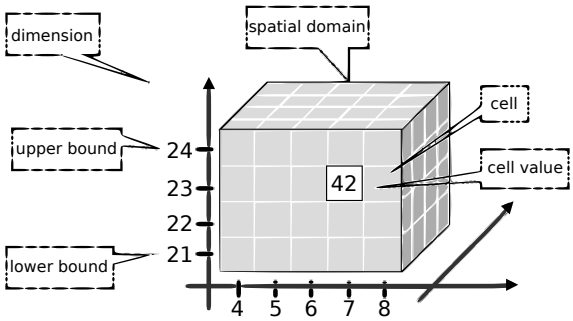

Figure 1.2 – Constituents of a multi-dimensional array.

In Listing 1 we propose some examples of the RasQL query language, so you get an immediate idea of how you can manipulate arrays with `rasdaman`.

For further guidance on the `rasdaman` query language refer to the available guide [rasdaman Gmbh, 2014] or browse `http://rasdaman.org/browser/manuals_and_examples/manuals/doc-guides`. For a more scientific introduction to `rasdaman`, refer to Baumann et al. [1997].

[1] `http://rasdaman.org/`

```
$ # usage
$ rasql
$ # which collections are stored?
$ rasql -q 'select m from RAS_COLLECTIONNAMES as m' --out string | grep Result
  Result object 1: mean_summer_airtemp
  Result object 2: mr
  Result object 3: rgb
  [...]
$ # dimensionality of a collection?
$ rasql -q 'select sdom(m) from mr as m' --out string | grep Result
  Result element 1: [0:255,0:210]
$ # get the 120th row of 'mr' (hex dump)
$ rasql -q 'select m[*:*,119] from mr as m' --out hex | grep Result
  Result object 1: 43 43 42 30 1c  8 23 1b 1c  d  1  a  0  [...]
$ # export 'mr' collection as a PNG image
$ rasql -q 'select encode(m, "png") from mr as m' --out file | grep Result
  Result object 1: going into file rasql_1.png...ok.
$ # reading the base type of a cell in an array
$ rasql -q 'select dbinfo(m) from mr as m' --out string | grep baseType
  "baseType": "marray <char, 2>",
$ # import a file into a rasdaman collection called 'test'
$ rasql -q 'create collection test GreySet' --user rasadmin --passwd rasadmin
$ rasql -q 'insert into test values decode($1)' -f rasql_1.png \
> --user rasadmin --passwd radmin
  [...] reading file rasql_1.png...ok
    constant 1: GMarray
    Oid.....................:
    Type Structure.........:
    Type Schema............: marray< char >
    Domain.................: [0:22623]
    Base Type Schema.......: char
    Base Type Length.......: 1
    Data format........... : Array
    Data size (bytes).... : 22624
  Executing insert query...ok
  rasql done.
$ # overlay a grey box on the 'test' collection
$ rasql -q 'update test as m set m[0:10,0:10] '\
> '            assign marray x in [0:10,0:10] values 127c' \
> --user rasadmin --passwd rasadmin
$ rasql -q 'select m[9:12,0] from test as m' --out hex | grep Result
  Result object 1: 7f 7f  0  0
$ echo $(( 16 * 7 + 15 ))
  127
$ # finally delete the 'test' collection
$ rasql -q 'drop collection test' --user rasadmin --passwd rasadmin
```

Listing 1 – Examples of RasQL commands on multidimensional arrays.

As previously said, arrays require ad-hoc metadata to map them to the world of real georeferenced phenomena. Indeed there are components of `rasdaman` which are devoted to the handling of the geospatial interface of `rasdaman` arrays:

rasgeo This application is used to ease the ingestion of georeferenced rasters into `rasdaman`, possibly stacked to compose 3D spatial or spatio-temporal cubes.

Petascope The OGC services Java servlet; it relies on its own database of metadata and exposes `rasdaman` array data to the outside world.

SECORE The companion of Petascope and official OGC resolver for Coordinate Reference Sys-

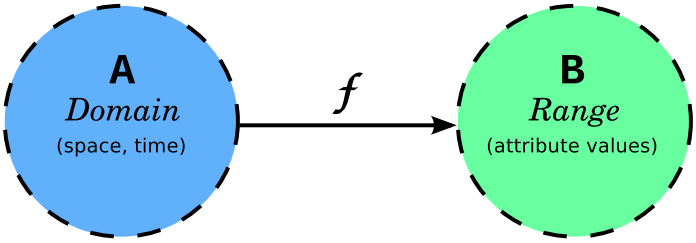

Figure 1.3 – A coverage is modelled as a function that returns values from its range for any direct position within its spatial, temporal or spatiotemporal domain.

tems (CRS); Petascope relies on this component to know all the semantics of the CRS space inside of which the coverages are defined.

In Chapter 2 we will learn how to use these components to serve coverages on the Internet. In the next section we will proceed with an explanation of the more advanced concept of *coverage*.

1.2 GMLCOV: the coverage model

The term "coverage" refers to any data representation that assigns values directly to spatial position: a coverage is a function from a spatial, temporal or spatiotemporal domain to an attribute range, as depicted in Figure 1.3. Coverages can include rasters, triangulated irregular networks, point clouds and polygon coverages, and they are the prevailing data structures in a number of application areas, such as remote sensing, meteorology and mapping of bathymetry, elevation, soil and vegetation [OGC, 2006].

A coverage *domain* consists of a collection of direct positions in a coordinate space that may be defined in terms of up to three spatial dimensions as well as one (or more, Baumann et al. 2012) temporal dimensions. A coverage *range* is the set of feature attribute values associated by a function with the elements of the domain.

In this document we will consider the so-called GMLCOV coverage model [Baumann, 2012], which is an extension to the core GML coverage model and which provides richer coverages by means of two additional elements:

- the `rangeType` element, which describes the coverage's range set data structure.
- the `metadata` element, to define concrete metadata structures and their semantics in extensions or application profiles.

Indeed, a range value often consists of one or more fields (in remote sensing also referred to as bands or channels), however, much more general definitions are possible. The `rangeType` additional element thus describes range value structure based on the comprehensive SWE Common DataRecord [Robin, 2011] model. A UML model of the GMLCOV coverage structure is reported in Figure 1.4.

So how are *rasters* represented in the coverage model? There are mainly two kinds of coverage that relate to `rasdaman` more closely:

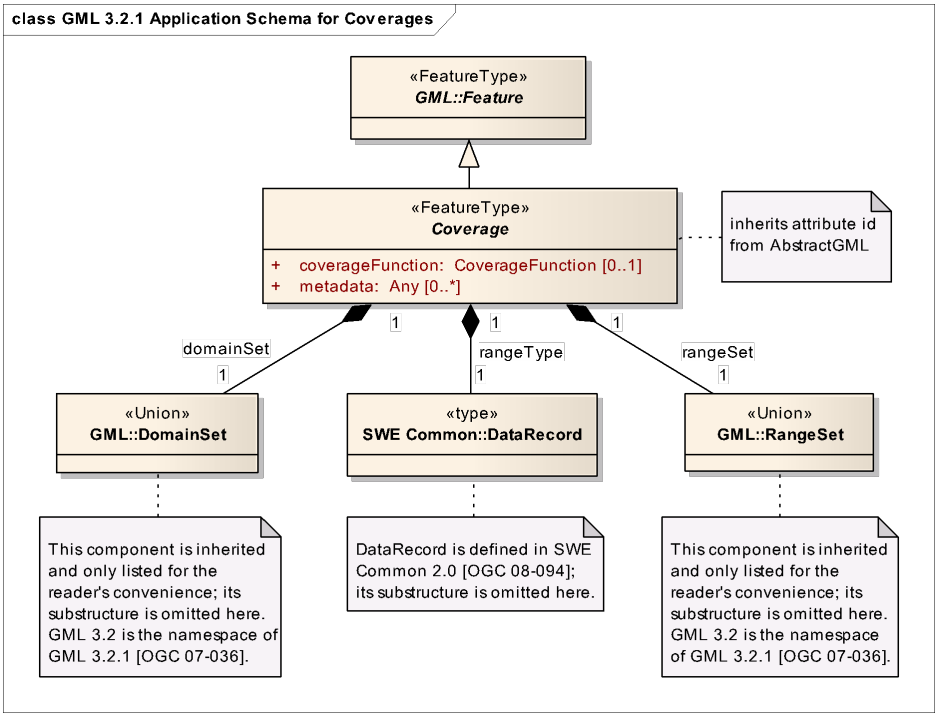

Figure 1.4 – The GMLCOV coverage structure.

`RectifiedGridCoverage` Coverage whose geometry is represented by a rectified grid. A grid is (geo)rectified when the transformation between *grid coordinates* and the external CRS is affine, like shifts, rotations and shearings; we also call them rectilinear aligned, or rectilinear non-aligned grids [Portele, 2007].

`ReferenceableGridCoverage` Coverage whose geometry is represented by a referenceable grid. A grid is (geo)referenceable when there is a non-affine transformation between the grid coordinates and the external CRS; this can be the case of rectilinear irreguarly-spaced grids, or curvilinear ("warped") grids [Portele, 2012].

Indeed, while we traditionally think of a grid as a classical aligned and orthogonal set of rectilinear lines (as in the example ① of Figure 1.5), the formal definition says it is a network of curves (*grid lines*) intersecting in a systematic way, forming *grid points* — at the intersections — and *grid cells* — at the interstices. This means that grid lines need not be straight and orthogonal.

In GML, there is a third type of grid which acts as the *internal* representation of any (geo)rectified or referenceable geometric grid, simply called `Grid`: in this case we have no *ex-*

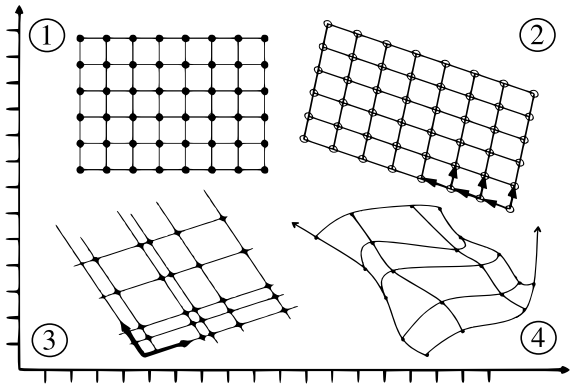

Figure 1.5 – Examples of 2D rectified (① and ②) and referenceable (③ and ④) grids.

ternal CRS coordinates, but integer indexing of the grid points along the orthogonal grid lines. `Grid`s are strictly related to the concept of marrays in `rasdaman`, which indeed are the internal representation of the geo-coverages exposed to the WCS service.

`rasdaman` supports the storage of rectified grids whose grid lines (or in this case *grid axes*) are *aligned* with the axes of the external cartesian coordinate system. However, when stacking rectified grids together in a same marray for building a spatial cube, or time series of images and cubes, irregular spacing is permitted[2] so that the overall coverage becomes referenceable (rectilinear, aligned and irregularly spaced).

Time is embedded in the geometry of a coverage by i) encoding time information through a temporal CRS which defines the epcoh time and the time step and ii) composing it with the usual geospatial projections [Campalani et al., 2013b]. More on time integration will be explained in Section 1.3.1.

A last important point of discussion concerns the sample space of grid points (which must not be confused with the grid cells between the grid interstices). The feature attribute values associated with a grid point represent characteristics of the real world measured within a small space surrounding a sample point: the *sample space*. The representation of a sample space in a CRS is called *footprint*. When dealing with gridded data, it is usually assumed (like we do) that the *sample cells* equally divided among the sample points so that they are represented by a second set of cells congruent to the *grid cells* but offset so that each has a grid point at its center [OGC, 2006].

Note that this assumption is not valid anymore on irregular grids where no inherent resolution exists, but this will be covered in more detail Section 1.3.1).

For more on coverages and grids, you can refer to the cited OGC normatives, or to Campalani et al. [2013a] for our usage of grids when building spatio-temporal coverages. In the next section we will describe the main features of the de-facto OGC standard for web services of coverages: the Web Coverage Service.

1.3 OGC Web Services on coverages

The reference OGC standard for publishing multi-dimensional coverages is the Web Coverage Service (WCS, `http://www.opengeospatial.org/standards/wcs`).

In the recent years, WCS has been completely overhauled to fulfill a more modular structure based on a core set of minimum requirements that a WCS-compliant service must adhere to, plus a plethora of extensions for additional service features, protocol bindings, format extensions and application profiles. This refactoring ended up in version 2.0 of the interface standard, the current version being 2.0.1 [Baumann, 2012].

WCS 2.0 offers several advantages over previous versions, like support for general n-D raster data and non-raster coverage types. It is also harmonized with GML and Sensor Web Enablement (SWE) models.

What are these cornerstone WCS functionalities then? As specifed in the WCS *core* standard, there are three kinds of request that can be sent to a WCS service:

[2]Starting from `rasdaman` version 9.0.0.

GetCapabilities This operation allows a client to request information about the server's capabilities and as well summary information on the offered coverages.

DescribeCoverage This operation allows a client to request a much more detailed description on the selected coverages, in particular their domain and range characteristics.

GetCoverage This operation allows a client to request a coverage *data*, usually expetided together with some of its metadata, depending on the selected output format[3].

As said before, many extensions can be plugged in a WCS service to add more interoperability and funcionalities. The most interesting (recently accepted) extensions are the following:

- ※ *Range Subsetting* — Baumann and Yu 2014a
  [Also available in `rasdaman` community] This extensions enables the selection of one or more attributes defined in the range of a coverage (e.g. extract arbitrary bands from an hyperspectral dataset).

- ※ *Scaling* — Baumann and Yu 2014b
  [Also available in `rasdaman` community] This extension makes it possible to upscale and downscale coverages via WCS requests.

- ※ *Interpolation* — Baumann and Yu 2014b
  [Also available in `rasdaman` community] The companion of the *Scaling* extension, it allows to declare the interpolation methods used when scaling (or when reprojecting, see *CRS* extension) and of course it allows to select a preferred interpolation in a *GetCoverage* request[4]

- ※ *Processing* — Baumann and Yu 2014
  [Also available in `rasdaman` community] The gate to the WCPS query language: this extension lets the user write arbitrary linear algebra expressions to be applied on coverages served by the WCS service.

- ※ *CRS* — Baumann and Yu 2014a
  [Available in `rasdaman` enterprise] This extension allows reprojection of both input WCS subsets — especially useful to let you subset a projected coverage via latitude/longitude degrees — and output coverage maps.

In the next chapter you will see many of these extensions in action. For more about the WCS service 2.0, refer to Baumann [2010a].

As mentioned in the beginning of this chapter, there is an other important standard related to coverages: the Web Coverage Processing Service (WCPS, `http://www.opengeospatial.org/standards/wcps`.

[3]The default output format is GML, which can provide the richest detail of metadata with respect to other popular binary formats like GeoTIFF or HDF. A prototype of the upcoming version 2.0 of the "GML in JPEG2000" OGC standard (`http://www.opengeospatial.org/standards/gmljp2`) is also available with `rasdaman`, in conjunction with GDAL v10.0 or later.

[4]`rasdaman` currently only implements the neareast-neighbor interpolation.

Although available in WCS via the *Processing* extension, the WCPS grammar is a standalone standard. WCPS brings to you the potential of a full array algebra that you can apply to one or more coverages together so that you can fetch the final product and forget about further client-side computations.

The general structure of a WCPS requests is given in Listing 2: the so-called "processing expression" is applied on each of the coverages specified in the given list (`coverageList_*`), given that the optional boolean expression returns `true` when evaluated on the coverage. Each coverage is referred to in the query by the correspondent identifier `variableName_*` in the processing expression.

```
for variableName_1 in ( coverageList_1 )
*[, variableName_N in ( coverageList_N ) ]
 [ where booleanScalarExpr ]
return processingExpr
```

Listing 2 – Template of a WCPS processing expression.

A processing expression (`processingExpr` in the listing above) branches down to a miscellanea of possible sub-expressions that are able to return either scalars (*scalar expressions*) or encoded marrays (*coverage expressions*) and operate on both the data and metadata of a coverage.

You will see some of the WCPS capabilities in the next chapter, but for a more comprehensive study on WCPS you can refer to the normative standard [Baumann, 2009] or as well the following selection of scientific publications: Baumann [2010b], Passmore [2013], Campalani et al. [2014]. Further useful links are available in Section 1.3.2 too.

1.3.1 Our WCS implementation at `rasdaman`

Some final comments go the WCS 2.0 implementation offered by `rasdaman` (also known as *Petascope*, see Figure 1.6).

Firstly, the service[5] follows the *minimal bounding-box* policy. This means that the input bounding-box requested by an enduser is adjusted to the footprints of the grid points.

Concerning the assumptions on the coverage sample spaces (see Section 1.2), our service behaves differently on each single coverage dimension:

→ for a *regularly-spaced* grid dimension, the footprint of the grid points along this axis are extended by half-resolution on both sides, having the grid point at the center of the sample space.

→ for an *irregularly-spaced* grid dimension — having no formal way to specify arbitrary sample spaces — the footprint of the grid points along this axis are 0-dimensional.

In practice, this has been conceived as the most intuitive behavior for this kind of service: this way the geographic extension of input geo-rasters is kept and each "pixel" of a raster is represented by a single grid point, positioned in its center. For irregular dimensions, it is foreseen to investigate solutions for specifying (and declaring) custom sample sizes.

[5]Starting from version 9.0.0.

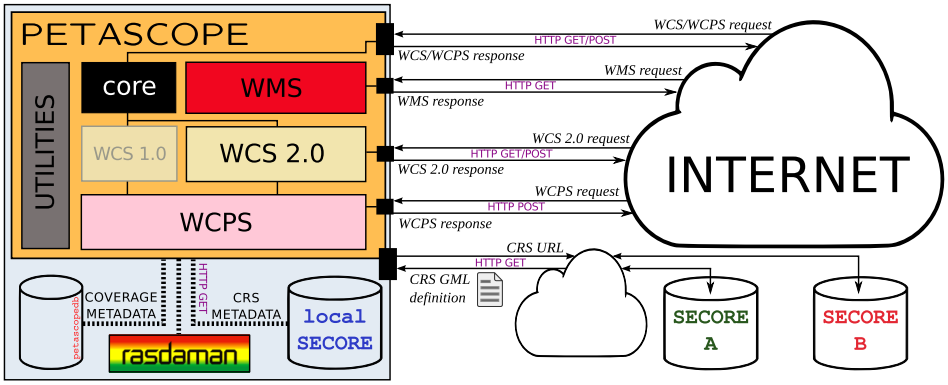

Figure 1.6 – Infrastructure of `rasdaman` and its *Petascope* and SECORE components for exposing arrays to OGC web services.

The `rasdaman` WCS service is also pioneering a novel approach to the handling of time-series of data through the definition of temporal CRSs[6].

Such CRSs encode time information based on a linear counting of time-steps from an epoch date. As for other CRS-related resources at OGC, GML-encoded definitions are publicly available via HTTP URL. Thanks to SECORE [Misev et al., 2012], arbitrary CRS compositions can be constructed, this way giving you the possibility to build a single space-time aquarium of analysis for your coverages.

This framework can embed time information in the geometry of a coverage, but on the other side it needs to index time to *temporal coordinates*. To ease the user experience, our service lets you also request WCS and WCPS subsets by replacing numeric time coordinates with human-readable ISO:8601 timestamps if enclosed by double-quotes. The service does the conversion for you.

Just as a short appendix, hereby we remind you the ISO format specification for datetime strings:

ISO:8601 datetime format

```
date-opt-time     = date-element ['T' [time-element] [offset]]
date-element      = std-date-element | ord-date-element | week-date-element
std-date-element  = yyyy ['-' MM ['-' dd]]
ord-date-element  = yyyy ['-' DDD]
week-date-element = xxxx '-W' ww ['-' e]
time-element      = HH [minute-element] | [fraction]
minute-element    = ':' mm [second-element] | [fraction]
second-element    = ':' ss [fraction]
fraction          = ('.' | ',') digit+
```

Any other information about the rasdaman community implementations can be found in our wiki at `http://rasdaman.org`.

[6] A first set of time CRS definitions and URIs has been recently accepted at OGC.

1.3.2 I want *more*

There is a set of useful online demonstrators, tutorials and teaching material available online for `rasdaman`-based web services that have spawned from the efforts of the EarthServer project (2011—2014, `http://earthserver.eu`).

There you can learn more about specific applications of WCS and WCPS services, how to use them and appreciate the advantages of server-side processing capabilities.

Here's the list:

- ◇ "Big Earth Data Standards" demonstrator, by Jacobs University Bremen and rasdaman Gmbh
  `http://www.earthlook.org/`
- ◇ WCS and WCPS "for dummies" video-tutorials, by MEEO Srl
  `http://earthserver.services.meeo.it/media-content/`
- ◇ WCPS guides by the Plymouth Marine Laboratory (PML)
  `http://earthserver.pml.ac.uk/portal/how_to/`
- ◇ PlanetServer: using WCPS for planetary science
  `http://planetserver.jacobs-university.de/`
- ◇ WCPS overview by the British Geological Survey (BGS)
  `http://earthserver.bgs.ac.uk/petascopeWCPS.html`
- ◇ xWCPS demo[7]
  `http://earthserver2.madgik.di.uoa.gr:8080/xWCPSApplication/`

More links to related documentation as well as lighthouse applications endpoints can be found in the project's wiki at `http://earthserver.eu/trac`.

[7] What is xWCPS anyway? See Perperis et al. 2013.

Chapter 2

Hands-on

After the walk-though on `rasdaman` and the OGC WCS and WCPS standards of the previous chapter, it is time now to start practicing on real datasets.

The chapter is divided in three sections, for three different use cases. Each section will show you how to properly ingest the dataset into `rasdaman`, and then you will be given some examples on how to access it with the WCS and WCPS standards.

Starting with a simple 2D multiband Landsat image in Section 2.1, we will then learn how to properly handle i) a regular time-series of images in Section 2.2 and ii) an irregular time-series of images in Section 2.3.

Here is the list of images we will use:

```
$DATASETS/
├── 2D_multiband_image/
│   └── [ 66M] N-32-50_ul_2000_s.tif
├── Regular/
│   ├── [ 10M] MOD_WVNearInfr_20120104_34.tif
│   ├── [ 14M] MOD_WVNearInfr_20120105_34.tif
│   ├── [ 11M] MOD_WVNearInfr_20120106_34.tif
│   ├── [9.7M] MOD_WVNearInfr_20120107_34.tif
│   ├── [ 13M] MOD_WVNearInfr_20120108_34.tif
│   └── [ 14M] MOD_WVNearInfr_20120109_34.tif
└── Irregular/
    ├── [ 28M] FSC_0.01deg_201102010750_201102011250_MOD_panEU_ENVEOV2.1.00.tif
    ├── [ 28M] FSC_0.01deg_201102020700_201102021200_MOD_panEU_ENVEOV2.1.00.tif
    ├── [ 28M] FSC_0.01deg_201102030740_201102031240_MOD_panEU_ENVEOV2.1.00.tif
    ├── [ 28M] FSC_0.01deg_201102040820_201102041150_MOD_panEU_ENVEOV2.1.00.tif
    ├── [ 28M] FSC_0.01deg_201102050725_201102051230_MOD_panEU_ENVEOV2.1.00.tif
    ├── [ 20M] FSC_0.01deg_201302200800_201302201255_MOD_panEU_ENVEOV2.1.00.tif
    ├── [ 20M] FSC_0.01deg_201302210705_201302211215_MOD_panEU_ENVEOV2.1.00.tif
    └── [ 20M] FSC_0.01deg_201302220750_201302221255_MOD_panEU_ENVEOV2.1.00.tif
```

```
└─ [ 20M] FSC_0.01deg_201302230700_201302231200_MOD_panEU_ENVEOV2.1.00.tif
└─ [ 20M] FSC_0.01deg_201302240735_201302241240_MOD_panEU_ENVEOV2.1.00.tif
└─ [ 20M] FSC_0.01deg_201302250735_201302251230_MOD_panEU_ENVEOV2.1.00.tif
```

We assume that you already have installed **rasdaman** in your system; otherwise refer to the public installation instrucions on our wiki at **http://rasdaman.org/wiki/Install**. We will also let **rasdaman** apply its default tiling to the data collections: ad-hoc tiling optimizations fall outside of the scope of this tutorial.

You will firstly need to check that all the **rasdaman** components for dealing with geospatial data are properly deployed and configured, in particular:

- ✓ *rasgeo* for the ingestion of coverages via the **rasimport** utility (and its companion **raserase**); the configuration file for *rasgeo* is usually placed in the **$HOME/.rasdaman/rasconnect** file; the user guide is at **http://rasdaman.org/wiki/RasgeoUserGuide**.
- ✓ *Petascope* for the WCS and WCPS interface; the configuration file is called **petascope.properties** and the user guide is at **http://rasdaman.org/wiki/PetascopeUserGuide**.
- ✓ SECORE for the database of CRS definitions required by *Petascope*; a user guide is available at **http://rasdaman.org/wiki/SecoreUserGuide**.

The reference version for this tutorial is **rasdaman** v9.0.2. To check your **rasdaman** version, you can use the RasQL **version()** function, as shown in Listing 3. [1].

```
$ rasql -q 'select version()' --out string | grep Result
  Result object 1: rasdaman v9.0.2 on x86_64-linux-gnu, [...]
```

Listing 3 – Getting the version of **rasdaman** installation.

Lsiting 4 shows the shell variables that will be used as handy system-independent shortcuts throughout the document:

```
$ # variables
$ export DATASETS=$HOME/Desktop/DATASETS
$ export WCS2_ENDPOINT='http://localhost:8080/rasdaman/ows/wcs2'
$ export WCPS_ENDPOINT='http://localhost:8080/rasdaman/ows/wcps'
$ export SECORE_ENDPOINT='http://localhost:8080/def'
$ # checks
$ rasimport
$ wget "${SECORE_ENDPOINT}/crs/OGC/0/"
$ wget "${WCS2_ENDPOINT}?service=WCS&version=2.0.1&request=GetCapabilities"
```

Listing 4 – Enviroment shell variables used in this chapter.

As explained in Section 1.3, the WCPS is a standalone OGC standard which can as well be embedded in a WCS query through the WCS 2.0 *Processing* extension. This means that you have two choices when sending the proposed WCPS queries[2]:

[1] RasQL '**version()**' function is not available for releases prior to v9.0.0.

[2] We will use HTTP GET requests via the Key/Value Pair (KVP) encoding, which is described in the WCS KVP encoding protocol binding [Baumann, 2013].

1. Percent-encode it as `query` parameter value in a WCS request of type `ProcessCoverages`;
2. Percent-encode it as `query` parameter value and ship it as body of an HTTP POST request to the WCPS servlet endpoint.

The WCPS servlet also provides a HTML landing page with forms that let you easily create, paste or upload your WCPS queries.

In conclusion, Listing 5 shows you some relevant information about the system and the software that were used to develop this tutorial:

```
$ uname -opsr
  Linux 3.2.0-64-generic x86_64 GNU/Linux
$ lsb_release -ds
  Ubuntu 12.04.4 LTS
$ bash --version | head -n1
  GNU bash, version 4.2.25(1)-release (x86_64-pc-linux-gnu)
$ psql --version
  psql (PostgreSQL) 9.3.4
$ python --version
  Python 2.7.3
```

Listing 5 – Information on the system used for this tutorial.

2.1 Single 2D image

In this section we will start by storing and querying a single rectified 2D image in `rasdaman`. After describing and analyzing the dataset, we will use *rasgeo* to import it in the database (Subsection 2.1.1), then we will show how to use WCS and WCPS to play with it (Subsection 2.1.2).

The Landsat program offers the longest continuous global record of the Earth's surface since 1972. It has moderate spatial resolution, can be used to many research area like disaster recovery, flood, city growth, etc. As a joint program of NASA and USGS, the Landsat archive is free available to everyone on everywhere on the Earth. This sample RGB image — which we will call *Multiband* — has been downloaded from USGS (`https://landsat.usgs.gov/`) and was taken by the Enhanced Thematic Mapper (ETM) sensor onboard Landsat7. It covers part of the North of Germany and Nord Sea, and was acquired from 2000 to 2001.

2.1.1 Data ingestion

Before importing the image, we need to know about its metadata: Listing 6 shows some relevant information about the structure and geometry of this dataset.

We can see that this is a 4657×4923 32N-UTM projected ($\rightarrow$EPSG:32632) RGB image, with resolution around 57 meters on both easting and northing directions, and origin in the upper-left corner[3].

Let's see now how to achieve a correct insertion of this image into `rasdaman`:

[3] `rasdaman` will always model a geospatial grid coverage with the origin in the upper-left corner anyway.

```
$ export IMAGE2D="${DATASETS}/2D_multiband_image/N-32-50_ul_2000_s.tif"
$ gdalinfo "$IMAGE2D" | grep 'Band'
  Band 1 Block=4657x1 Type=Byte, ColorInterp=Red
  Band 2 Block=4657x1 Type=Byte, ColorInterp=Green
  Band 3 Block=4657x1 Type=Byte, ColorInterp=Blue
$ gdalinfo "$IMAGE2D" | grep '^Size'
  Size is 4657, 4923
$ gdalinfo "$IMAGE2D" | grep 'PROJCS'
  PROJCS["WGS 84 / UTM zone 32N",
$ gdalinfo "$IMAGE2D" | grep 'Origin' -A1
  Origin = (234989.621940090029966,6097439.627894577570260)
  Pixel Size = (57.006119819626370,-57.005789152955515)
```

Listing 6 – Metadata for the 2D image sample dataset.

```
$ # data type
$ rasdl -p | grep RGBImage
  typedef marray <struct { char red, char green, char blue }, 2> RGBImage;
  typedef set <RGBImage> RGBSet;
$ # ingest
$ rasimport -f "$IMAGE2D" \
> --coll 'Multiband' \
> --coverage-name 'Multiband' \
> -t RGBImage:RGBSet \
> --crs-uri '%SECORE_URL%/crs/EPSG/0/32632'
$ # check
$ rasql -q "select sdom(m) from Multiband as m" --out string | grep Result
  Result element 1: [0:4656,0:4922]
$ rasql -q "select m[2000,1000] from Multiband as m" --out string | grep Result
  Result element 1: { 11, 6, 15 }
$ python
>>> import Image, numpy, os
>>> numpy.array(Image.open(os.environ['IMAGE2D']))[1000,2000]
    array([11,  6, 15], dtype=uint8)
$ wget "${WCS2_ENDPOINT}?service=WCS&version=2.0.1&"\
> "request=DescribeCoverage&"\
> "coverageid=Multiband"
```

Listing 7 – Ingesting the RGB 2D image sample dataset.

In Listing 7 we first verify the data type that we need to assign and that `rasdaman` can understand, and that is an `RGBSet`, which in fact is defined as a set of `RGBImage` elements, which in turn are a 2D array of RGB structs, just like our Landsat image.

If you wonder why `RGBImage` is not enough as a data type, remember from Section 1.1 that arrays in `rasdaman` are encapsulated inside *collections*, which can contain one or more arrays. A *coverage* is identified by a single marray, however the *collection* concept might be used for grouping coverages together in a same logical bag, as we are going to show later in this section. To remind you that coverages and collec-

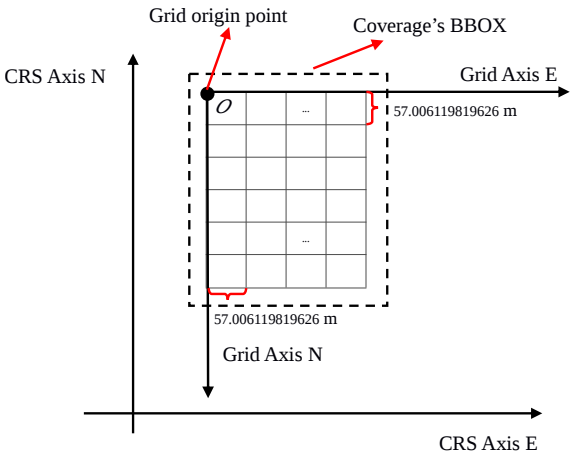

Figure 2.1 – Visualization of the domain of the imported *Multiband* grid coverage with origin in the upper-left corner and bounding-box (dashed line) given by the grid sample cells, and not by the grid points.

```
$ # rasimport synopsis
$ rasimport | grep '^ --bnd'
    --bnd     spatial import boundary (i.e. sub-setting import file(s))
              (e.g. xmin:xmax:ymin:ymax)
$ # ingest
$ rasimport -f "$IMAGE2D" \
> --coll 'Multiband' \
> --coverage-name 'MultibandPart' \
> -t RGBImage:RGBSet \
> --crs-uri '%SECORE_URL%/crs/EPSG/0/32632' \
> --bnd 236000:237000:5850000:5851000
$ # check
$ rasql -q "select sdom(m) from Multiband as m" --out string | grep Result
  Result element 1: [0:4656,0:4922]
  Result element 2: [0:16,0:17]
$ wget "${WCS2_ENDPOINT}?service=WCS&version=2.0.1&"\
> "request=DescribeCoverage&"\
> "coverageid=MultibandPart"
```

Listing 8 – Ingesting a spatial subset of the 2D image sample dataset.

tions are different animals, you need to specify two separate labels during ingestion: one for the collection (`--coll`) and one for the coverage (`--coverage-name`), in our case both set to *Multiband*.

Thanks to the high-level interface of *rasgeo*, the data type was actually the only information we had to fetch for importing the Landsat image. Due to our URI-based handling of coordinate refence systems through SECORE, the only additional metadata which *rasgeo* cannot parse from the image header is the CRS identifier, which we indeed set as `--crs-uri` parameter in the `rasimport` call.

The `%SECORE_URL%` keyword is our shortcut for a more portable ingestion: *Petascope* will replace this keyword with the actual SECORE endpoint configured in its `petascope.properties` file. There is no need to use this keyword, nor to use a single SECORE host anyway: our concept, as it was depicted in Figure 1.6, is to have a distributed family of SECORE resolvers over the Internet, where each one is free to offer some CRSs definitions you might want to use for your data.

As `rasimport` terminates, you already have transferred the Landsat image in the database. Listing 7 then just proceeds with some validity check using comparing RasQL with python pixel selection and finally verifies that the coverage — sketched in Figure 2.1 — is already registered in the WCS server by means of a *DescribeCoverage* request. Indeed, the coverage is published instantly in the server after ingestion.

With `rasimport` it is also possible to import only a subset of the input image by passing a bounding box as '`--bnd`' argument, as shown in Listing 8. We will call the coverage *MultibandPart* and we will put it inside the same collection *Multiband* created before, to underline the logical association of these two arrays. Again, validity checks and a WCS *DescribeCoverage* request are done after ingestion.

Now that the Landsat image is correctly imported in `rasdaman`, we will see some possible access and processing use-cases that can done via WCS and WCPS requests.

As a last note, while RGB 3-byte structs is already available as built-in type in the database, you can always use the `rasdaman` Definition Language (RasDL) to create your own base data type like for your multi/hyper spectral rasters or for estimation/error model maps, for instance.

2.1.2 WCS in action

We already verified in the previous section that our *Multiband* coverage has been registered in the WCS (and consequently WCPS) service, so now we can start sending some *GetCoverage* request to our server.

As a first basic example, we might just want to retrieve a spatial cutout of the coverage by using `subset` KV-pairs as shown in Listing 9:

```
${WCS2_ENDPOINT}?
  service=WCS&
  version=2.0.1&
  request=GetCoverage&
  coverageId=Multiband&
  subset=E(300000,370000)&
  subset=N(5800000,5850000)&
  format=image/tiff
```

Listing 9 – Fetching a subset image from *Multiband* coverage via WCS. The response is shown in Figure 2.2.

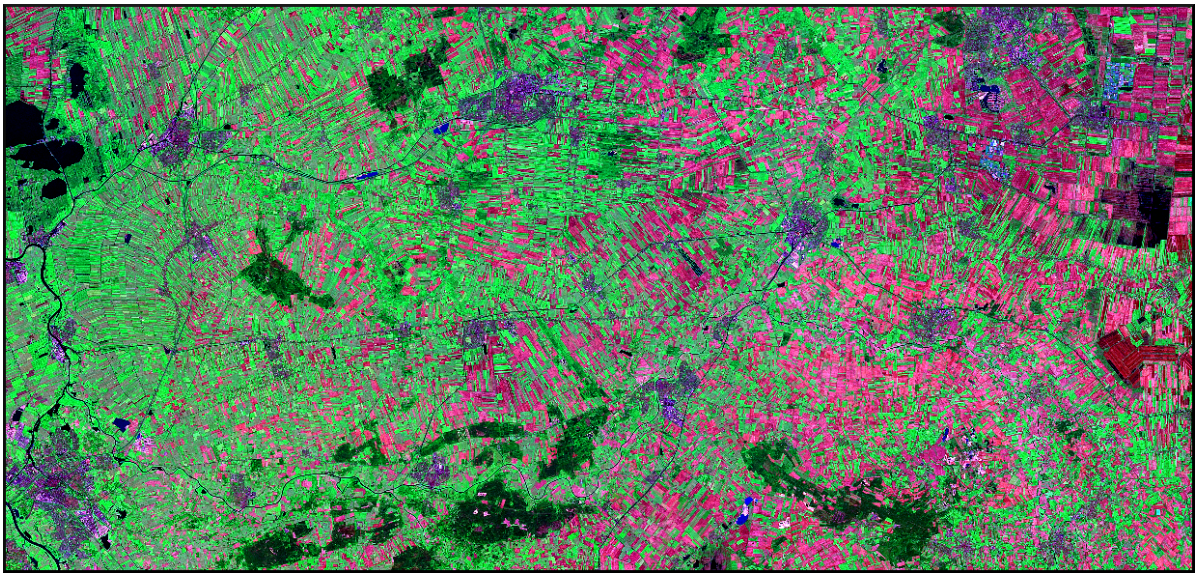

Figure 2.2 – Output of the WCS *GetCoverage* request from Listing. 9 : spatial subsetting.

As you see, we were able to specify our region of interest by specifying intervals on separate CRS axes. The labels that we used to identify a subset dimension are strictly equal to the labels (`gml:axisAbbrev`) declared in the definition of the CRS, as shown in Listing 10:

The general usage of a KV-encoded WCS subset is then:

$$\text{subset} = label_{axis}(coord_{lo}[, coord_{hi}])$$

When a single direct position is specified in a subset, then the coordinate is usually referred to as *slicing position*, and the subset is called a *slice*; otherwise the subsetting operation is also called *trimming*.

```
$ wget "${SECORE_ENDPOINT}/crs/EPSG/0/32632" -qO - | \
> grep '<gml:axisAbbrev>' | tail -n 2
  <gml:axisAbbrev>E</gml:axisAbbrev>
  <gml:axisAbbrev>N</gml:axisAbbrev>
$ # (use 'tail' to exclude the labels from the base geographic CRS)
```

Listing 10 – Reading the CRS axis labels from the CRS definition stored in SECORE.

☛ It is highly important to understand that slicing a CRS dimension automatically decreases the dimensionality of the CRS. Due to the 1:1 association between a CRS axis and a grid coverage axis in our WCS implementation, this also means that the grid (array) dimensionality gets reduced.

You also can see again from Listing 9 that we explictly specified `image/tiff` (that is a GeoTIFF in our case) as output format for our response. By default our service returns GML-encoded responses otherwise: while GML can give you the highest richness of metadata, it is surely more convenient to retrieve cheaper binary formats when the data to be downloaded is sufficiently big.

As a second example, we can now use the *Range Subsetting* WCS extension to extract a single band from the RGB coverage. This can be achieved by adding a `rangesubset` KV-pair in the request, as shown in Listing 11:

```
${WCS2_ENDPOINT}?
  service=WCS&
  version=2.0.1&
  request=GetCoverage&
  coverageId=Multiband&
  subset=E(300000,370000)&
  subset=N(5800000,5850000)&
  rangesubset=b1&
  format=image/tiff
```

Listing 11 – Fetching a subset image from *Multiband* coverage via WCS. The response is shown in Fig. 2.3.

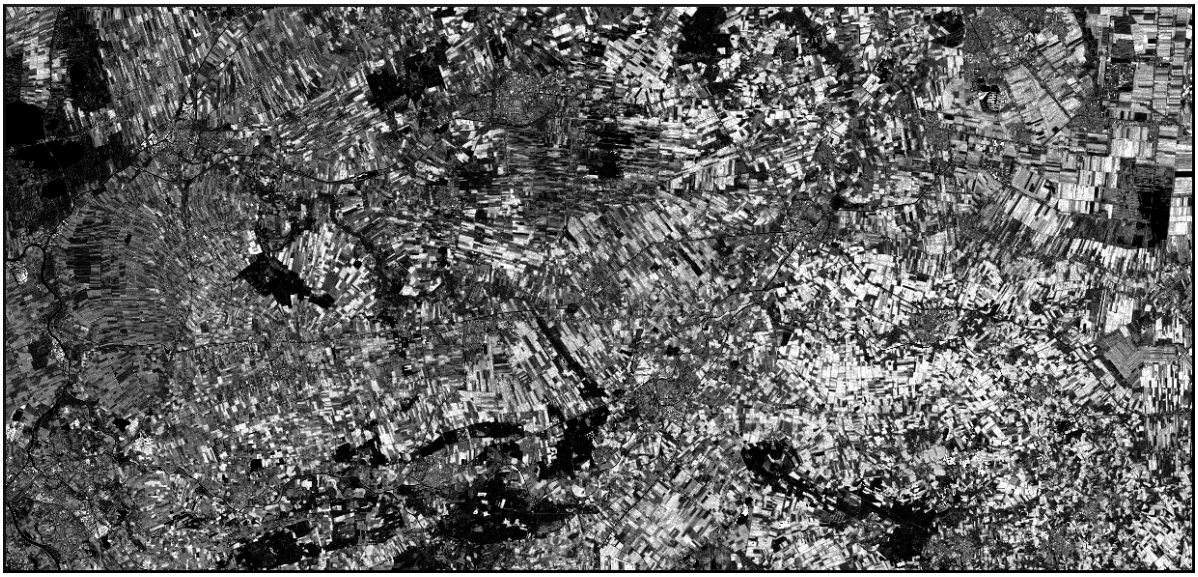

Figure 2.3 – Output of the WCS *GetCoverage* request from Listing. 11 : extracting the red channel of a spatial subset from an RGB image.

```
$ wget "${WCS2_ENDPOINT}?${WCS2_COMMON_KVP}&"\
> "request=DescribeCoverage&"\
> "coverageid=Multiband" -qO - | grep 'field name'
   <swe:field name="b1">
   <swe:field name="b2">
   <swe:field name="b3">
```

Listing 12 – Reading the coverage attribute names from the SWE record (*range type*) of its coverage description. This information is also available in a *GetCoverage* response.

The label of the coverage attribute (`b1`) has been automatically set by *rasgeo* while importing and you can verify it by going through the GML coverage description, as shown in Listing 12. You currently have to tweak the metadata database to customize your SWE record: not only you can change the label, but you can add further descriptive information, definition URIs, allowed values and declare multiple Nil values and reasons (see our developer's guide at `http://rasdaman.org/wiki/PetascopeDevGuide`).

In addition to the exctraction of a single band, the WCS *Range Subsetting* extension gives you some more flexibility, by letting you ask for:

- separate bands: $< B_\alpha, B_\beta, B_\gamma, \ldots >$
  for instance '`rangesubset=b1,b3`' to extract the red and blue channels
- a range of bands: $< B_{from} : B_{to} >$
  for instance '`rangesubset=b1:b2`' to extract the red and (to) green channels

While WCS is much about raw data access, you can go beyond simple READ operations with the WCPS query language.

Spectral indexes like NDVI and the like are usually the very first example of things that WCPS can do quite easily. Range subsetting is natively available in WCPS by means of a dot '`.`' notation; trimmings and slicings are — of course — available too, but beware that the coordinates separator for a trim subset is here the colon '`:`' (and not the comma '`,`' as in a WCS subset).

In the following example (Listing 13) we retrieve a simple spectral index given by the average of the blue and the red bands.

```
for cov in (Multiband)
return encode(
   ((cov.b3+cov.b1)/2)[E(490000:492000),N(6000000)],
"csv")
--------
{113,77,54.5,77,0.5,85.5,107.5,110, ... ,122,74.5}
```

Listing 13 – Getting average pixels of red and blue bands of the *Multiband* coverage on a selected area of interest using WCPS.

You see that we specified here a `csv` (Comma-Separated Values) encoding for our processing expression (remember Listing 2): indeed we have *sliced* the `Northing` CRS axis on 6000000 meters North, so we could not ask for a GeoTIFF response in this case: our output is here 1D, whereas image formats require a 2-dimensional dataset.

An other useful application of WCPS is to build false-color images from a multi- or hyper-spectral image. Although our sample image has only three channels, we can show how to build the proper query by, for instance, changing the order of the RGB channels:

```
for cov in (Multiband)
return encode({
     red:   cov.b3;
     green: cov.b1;
     blue:  cov.b2
   }[E(300000:370000),N(5800000:5850000)],
"tiff")
```

Listing 14 – Creating a false-color image by exchanging the position of RGB bands in the *Multiband* coverage using WCPS. The response is shown in Figure 2.4.

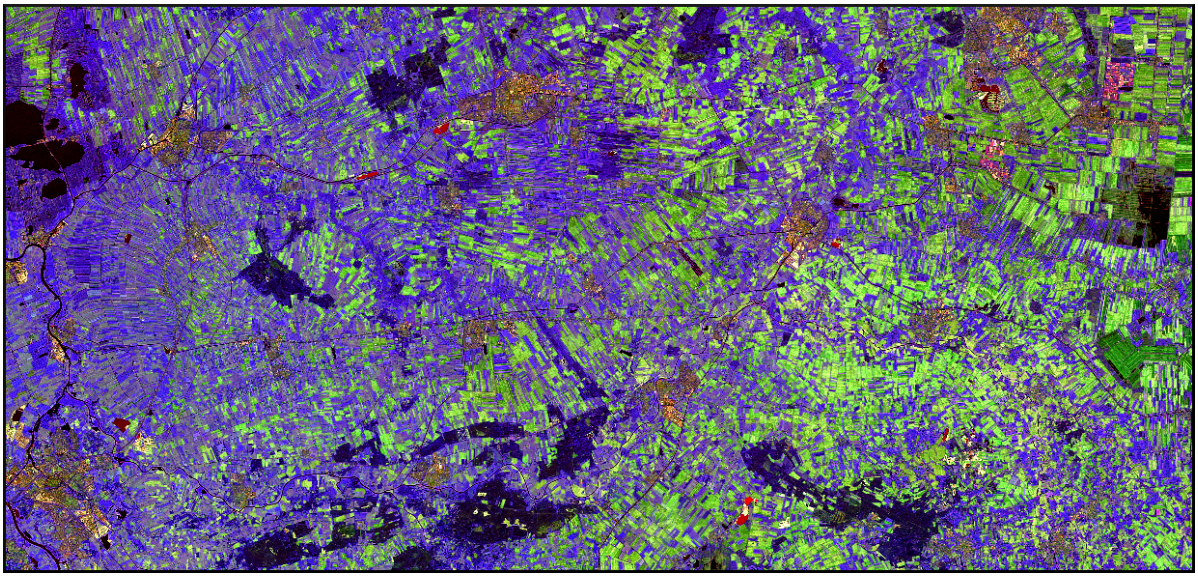

Figure 2.4 – Output of the WCPS request from Listing. 14 : creating a false-color image.

By using the proposed RGB WCPS constructor you can also specify a pixel-wise transparency by assigning an additional 'alpha' channel to some other coverage field or maybe some predefined mask coverage you can add in the input coverage list.

Hoping that everything is *fine*, we now go on to the next section and start handling multiple images of a same product at different time instants and publishing them as a single spatio-temporal coverage.

2.2 Regular time-series of images

So far we learned how to import and retrieve a single geospatial image. In this section we will start handling a spatio-*temporal* dataset and hence we will need to take care now about how to integrate time information in our coverage domain.

As in the previous section, after describing and analyzing the dataset, we will use *rasgeo* to import

it in the database (Subsection 2.2.1), then we will show how to use WCS and WCPS to play with it (Subsection 2.2.2).

Our dataset is a regular time-series of daily MODIS-based Level 2 maps of water vapor aerosols. Aerosols are one of the greatest sources of uncertainty in climate modeling: they vary continuously in time and in space and can lead to variations in cloud microphysics, impacting cloud radiative properties and climate.

The MODIS Aerosol Product monitors the ambient aerosol optical thickness over the oceans globally and over a portion of the continents. The product is used to study aerosol climatology, sources and sinks of specific aerosol types (e.g. sulfates and biomass-burning aerosol), interaction of aerosols with clouds, and atmospheric corrections of remotely sensed surface reflectance over the land.

The dataset sample has been kindly provided by the Meteorological Environmental Earth Observation company (MEEO Srl — `http://www.meeo.it`)

2.2.1 Data ingestion

Again, before we start to import any data, we need to learn about it. Listing 15 shows the relevant geospatial information of a sample image: we can see that the dataset is a 620×4414 single band image of floating-point numbers, with resolution of 1 km and UTM projected (zone 34N $\rightarrow$EPSG:32634). We also know from the data providers that unavailable data pixels are set to -9999.

```
$ export REGULAR3D="${DATASETS}/Regular"
$ export REGULAR3D_SAMPLE="${REGULAR3D}/MOD_WVNearInfr_20120104_34.tif"
$ gdalinfo "$REGULAR3D_SAMPLE" | grep 'Band'
  Band 1 Block=620x3 Type=Float32, ColorInterp=Gray
$ gdalinfo "$REGULAR3D_SAMPLE" | grep '^Size'
  Size is 620, 4414
$ gdalinfo "$REGULAR3D_SAMPLE" | grep 'PROJCS'
  PROJCS["WGS 84 / UTM zone 34N",
$ gdalinfo "$REGULAR3D_SAMPLE" | grep 'Origin' -A1
  Origin = (190000.000000000000000,6789000.000000000000000)
  Pixel Size = (1000.000000000000000,-1000.000000000000000)
```

Listing 15 – Metadata for the regular 3D time-series sample dataset.

Before importing we now also have to know *when* is the data. As shown in Listing 16, our six daily images begin on January 4^{th} and end on January 9^{th}, 2012. We want to load this time-series as a regular sequence of images, where each slice occupies the 24 hours of one day: geometrically, we are going to place the grid points at noon, with a footprint of ± 12 hours along the time axis.

We need to select a temporal CRS for our time dimension, and we now decide to index time with their correspondent ANSI date numbers: the linear counting of days starting from January 1^{st} 1601 (day 1).

You can ask SECORE about the available time CRSs at `$SECORE_ENDPOINT/def/crs/0/OGC/`. You can always fetch the definition of a CRS in case you are not sure about its meaning. As of our ANSI date CRS, we can go trough its definition at `$SECORE_ENDPOINT/crs/OGC/0/AnsiDate` as shown in Listing 16. We quickly compute the ANSI date numbers that correspond to our dates of interest,

so we discover that we are in the range [150118, 150123].

```
$ # ANSI date CRS is days from 1601-01-01 (date 1), 'ansi' labelled:
$ wget "${SECORE_ENDPOINT}/crs/OGC/0/AnsiDate" -qO - | \
> egrep 'uom=|<axisAbbrev>|<origin>'
  <CoordinateSystemAxis id="day" uom="http://www.opengis.net/def/uom/UCUM/0/d">
  <axisAbbrev>ansi</axisAbbrev>
  <origin>1600-12-31T00:00:00Z</origin>
$ # ANSI/UNIX epoch delta is 134774 days
$ date -ud '1601-01-01 + 134774 days' +%F
  1970-01-01
$ # date to ANSI date number (1601-01-01 is day 1)
$ for day in $( ls "$REGULAR3D" | awk -F '_' '{ print $3 }' ); do
> echo $( date -ud "$day" +%F ) = \
> $(( $( date -ud "$day" +%s ) / (3600 * 24) + 134774 + 1 )) ANSI date
> done
  2012-01-04 = 150118 ANSI date
  2012-01-05 = 150119 ANSI date
  2012-01-06 = 150120 ANSI date
  2012-01-07 = 150121 ANSI date
  2012-01-08 = 150122 ANSI date
  2012-01-09 = 150123 ANSI date
```

Listing 16 – Getting ANSI date numbers of the *aerosol* regular time-series.

While `rasgeo` can parse the geo-information of images from their header on itw own, we have to instruct him on every aspect of time. As you can see in Listing 17, we can do so by specifiying the following additional arguments to `rasimport`:

`--3D`
: We set this flag to instruct *rasgeo* that this is going to be a 3-dimensional cube of images.

`--csz 1`
: Here we specify the *spacing* of each cell along the Z axis[4], which in our case is 1. Why is it 1? Because we set a CRS with *day* (`http://www.opengis.net/def/uom/UCUM/0/d`) as Unit of Measure (UoM) (see again Listing 16): so we are saying that each image is separated by 1 day.

`--shift 0:0:150118`
: We can specify here the X:Y:Z offsets that *rasgeo* can assign to a data cube: while we do not need any shift in the spatial dimensions, we tell `rasimport` that the temporal origin of our series is on the ANSI date number 150118. In this case 150118 represents the datetime January 4$^{\text{th}}$ 2012 00:00:00.000Z: the shift is interpreted by `rasimport` as the lower-bound of the associated origin sample cell, so you will automatically get the coverage origin at noon ($150118 + \frac{1}{2} = 150118.5$) of the same day.

As in the previous example in Section 2.1.1, we also specify the CRS URI identifier with the `--crs-uri` parameter, which in this case is a (colon-separated) *concatenation* of the UTM projection and our ANSI date CRS. The order of CRS compounding is important here, since this defines the order of spatio-temporal coordinates in the tuples of direct positions of the coverage's domain. The

[4] In the *rasgeo* nomenclature, the three axis of a cube are always referred to as X, Y and Z axes, independently of the CRS labels and of the kind of axes of the specific dataset.

order of correspondent array axes can differ from this CRS compounding (especially when using the (in)glorious EPSG:4326 or other 2D geographic CRSs which define latitude first), but this is no problem as `rasimport` provides an other parameter `-crs-order` for adjusting axes orders (see an example in the next section).

```
$ # data type
$ rasdl -p | grep FloatCube | grep 3
  typedef marray <float, 3> FloatCube;
  typedef set <FloatCube> FloatSet3;
$ # rasimport help
$ rasimport | egrep '^ --csz|^ --3D|^ --shift'
  --3D     mode for 2D (x/y) image slice import into 3D cubes
  --csz    z-axis cell size
  --shift  shifts the origin of the import image by the specified vector
$ # ingest
$ rasimport -d "$REGULAR3D" -s 'tif' \
> --coll 'aerosol' \
> --coverage-name 'aerosol' \
> -t FloatCube:FloatSet3 \
> --crs-uri '%SECORE_URL%/crs/EPSG/0/32634':'%SECORE_URL%/crs/OGC/0/AnsiDate' \
> --csz 1 \
> --3D top \
> --shift 0:0:150118
$ # check
$ rasql -q "select sdom(m) from aerosol as m" --out string | grep Result
  Result element 1: [-14:633,-700:5322,150117:150122]
$ rasql -q "select m[300,2000,150118] from aerosol as m" \
> --out string | grep Result
  Result element 1: 0.611
$ python
>>> import Image, numpy, os
>>> numpy.array(Image.open(os.environ['REGULAR3D_SAMPLE']))[2000,300]
    0.611
$ wget "${WCS2_ENDPOINT}?service=WCS&version=2.0.1&"\
> "request=DescribeCoverage&"\
> "coverageid=aerosol" -O describeAerosols3D.xml
$ grep Corner describeAerosols3D.xml
  <lowerCorner>176000 1466000 150118</lowerCorner>
  <upperCorner>824000 7489000 150124</upperCorner>
$ grep -o 'axisLabels=.* ' describeAerosols3D.xml | head -n 1
  axisLabels="E N ansi" uomLabels="metre metre d"
```

Listing 17 – Ingesting the regular 3D MODIS time-series.

As you see from the RasQL `sdom` response, the marray index for the third (temporal) dimension got shifted to 150118 ANSI date number but this is just done for convenience by the *rasgeo* component, since – remember – collections are totally unaware of real-world semantics.

Concerning time, again: you see that the bounding box shown in the WCS coverage description indeed goes from January 4th 2012 00:00:00 (150118 ANSI date) to January 10th 2012 00:00:00 (150124 ANSI date): our last image of the series spans in fact the full January 9th day, so that at the upper-bound border of this cell in time we reach January 10th 2012 00:00:00.

Curiously, you see in Listing 17 that the marray indexes for the are not exactly on the usual default 0 origin: this happened because actually the time-slices in our regular time-series — which by the way we have called *aerosol* — are not exactly co-located on the projection plane, so *rasgeo* had to manually shift each new slice so that they all get overlaid properly in the UTM projection. This is one of the great advantages of using *rasgeo*.

Indeed, looking at Listing 18, you can see the different bounding boxes of each time slice: the overall bounding box in the cube indeed encloses all the images (see *DescribeCoverage* response in Listing 17)

```
$ for image in $( find $REGULAR3D -name "*.tif" ); do
> echo :: $( basename "$image" )
> gdalinfo "$image" | grep Lower\ Left -A1
> done
  :: MOD_WVNearInfr_20120104_34.tif
  Lower Left  (  190000.000, 2375000.000) ( 18d 0'32.82"E, 21d27' 2.85"N)
  Upper Right (  810000.000, 6789000.000) ( 26d45'25.93"E, 61d 6'45.91"N)
  :: MOD_WVNearInfr_20120105_34.tif
  Lower Left  (  176000.000, 1466000.000) ( 18d 0'37.77"E, 13d14'36.55"N)
  Upper Right (  824000.000, 6942000.000) ( 27d17'22.23"E, 62d28' 1.27"N)
  :: MOD_WVNearInfr_20120109_34.tif
  Lower Left  (  179000.000, 1690000.000) ( 18d 0'41.77"E, 15d15'59.14"N)
  Upper Right (  822000.000, 7190000.000) ( 27d45'27.64"E, 64d40'48.52"N)
  :: MOD_WVNearInfr_20120108_34.tif
  Lower Left  (  179000.000, 1759000.000) ( 18d 0' 9.28"E, 15d53'21.78"N)
  Upper Right (  807000.000, 7147000.000) ( 27d21'18.97"E, 64d18'38.45"N)
  :: MOD_WVNearInfr_20120107_34.tif
  Lower Left  (  196000.000, 2626000.000) ( 18d 1' 6.82"E, 23d42'58.65"N)
  Upper Right (  805000.000, 6794000.000) ( 26d40'22.50"E, 61d 9'40.80"N)
  :: MOD_WVNearInfr_20120106_34.tif
  Lower Left  (  196000.000, 2686000.000) ( 18d 0'21.82"E, 24d15'26.97"N)
  Upper Right (  784000.000, 7489000.000) ( 27d37'38.73"E, 67d22'50.38"N)
```

Listing 18 – Different bounding-boxes for each of the images in the *aerosol* time series, automatically handled by *rasgeo* during ingestion.

We have now understood all the mechanics behind the ingestion of our ANSI-date encoded time-series of MODIS images as a 3-dimensional coverage. In the next section we try to fetch some interesting information from it via WCS and WCPS queries.

2.2.2 WCS in action

The very first thing we can do now that we have a (regular) time-series of images is clearly to go *through* time. Many platforms today provide the *so-called* pixel histories to depict time profiles of certain products over a single location (pixel) or by selecting an area of interest [Natali et al., 2011].

This is easily done via a core WCS query by using slicing the easting and northing dimensions over a single point, then trimming time to select an interval of interest (or otherwise just do not subset it at all to keep all the available depth in time). The query is presented in Listing 19.

```
${WCS2_ENDPOINT}?
  service=WCS&
  version=2.0.1&
  request=GetCoverage&
  coverageId=aerosol&
  subset=E(500000)&
  subset=N(4000000)&
  subset=ansi(*,*)
```

Listing 19 – Time history over a single pixel location on *aerosol* coverage via WCS.

What you will receive is a GMLCOV-encoded 1D array of aerosols values over (500000E, 4000000N) [m]

and spanning all the available days of our time series.

As a second example, we now want to extract an Easting/time Hovmöller diagram from our cube. Again to do so we just node the core WCS capabilities by properly orchestrating the subsets type and extent on the three CRS axes that we have. This is shown in Listing 20

```
${WCS2_ENDPOINT}?
  service=WCS&
  version=2.0.1&
  request=GetCoverage&
  coverageId=aerosol&
  subset=N(4000000)&
  subset=E(500125,510975)&
  subset=ansi("2012-01-01","2012-01-31")
```

Listing 20 – Hovmöller Easting/time map on *aerosol* coverage via WCS.

Has previously mentioned (Section 1.3.1), while you actually encoded time with numerical coordinates during ingestion, you can retrieve your data using human-readable ISO:8601 timestamps, when enclosed by double-quotes (WCS core standard recommendations, Baumann 2012), Nonetheless you can alway directly ask for an ANSI date number: for instance you can interchangeable use these two slicing subsets: `ansi("2012-01-04")` or `ansi(150118)`.

If you look at the envelope of the returned coverage (by just running the query in Listing 20 in your browser or from the terminal), you see that the *minimal* bounding box is returned: while we requested the values between 500125 and 510975 metres of eastings, the returned data have been adjusted to the interval $(500000, 511000)$ instead, respecting the coverage resolution of 1 km.

A last example now on possible use-cases for a WCPS query. While there is a Big Variety of possible applications of WCPS to these kinds of data [Campalani et al., 2014], we will propose here a simple but yet delicate example: calculating the mean value of a selected time-slice.

WCPS provides the `avg` operator to compute the average over a multi-dimensional array, so one could very simply run the WCPS request proposed in Listing 21.

```
for cov in (aerosol)
return encode((float)
   avg(cov[ansi("2012-01-08")]),
"csv")
--------
{-3400.09}
```

Listing 21 – Getting the average of the map on 2012-01-08 of the *aerosol* coverage using WCPS.

You can see however that some nil values (-9999) in the time-slice have counterfeited the mean concentration (optical depths) of aerosols, whose values are instead close to 0 on average (but we received a value close to -3400!).

So we need to find a way to exclude nil values from our computations[5], and to do so we basically rewrite the average as the explicit sum of the elements, divided by the their cardinality ($\sum_i^n \frac{x_i}{n}$) then we manually trim down to 0 the nil -9999 pixels with the help of boolean logic, as exposed in Listing 22: as you see, we now have a correct average value in response (0.672708).

[5]The enterprise version of `rasdaman` provides seamless nil handling instead.

```
for cov in (aerosol)
return encode((float)
   add(
      (cov[ansi("2012-01-08")]  = -9999) * 0 +
      (cov[ansi("2012-01-08")] != -9999) *
       cov[ansi("2012-01-08")]
  ) / count(cov[ansi("2012-01-08")] != -9999),
"csv")
--------
{0.672708}
```

Listing 22 – Getting the average of the map on January 8th 2012 of the *aerosol* coverage using WCPS, excluding nil values.

Again, hoping everything is still *fine*, we now proceed to the last section of the tutorial to learn how to import an irregular time-series of images into `rasdaman`.

2.3 Irregular time-series of images

In this final section we are going to ingest an irregular time-series of images.

As in the previous sections, after describing and analyzing the dataset, we will use *rasgeo* to import it in the database (Subsection 2.3.1), then we will show how to use WCS and WCPS to play with it (Subsection 2.3.2).

The images that we will use represent pan-European fractional snow cover products that hydrologists can use for estimating the water mass available in a river basin, for instance. The data has been kindly made available by "Cryoland" Copernicus service, which provides geospatial products on the seasonal snow cover, glaciers, and lake/river ice derived from Earth observation satellites by means of geo-portals (`http://cryoland.eu/`).

2.3.1 Data ingestion

As usual, we will start by skimming through the information that GDAL can parse from the files (Listing 23).

```
$ export IRREGULAR3D="${DATASETS}/Irregular"
$ export IRREGULAR3D_SAMPLE="${IRREGULAR3D}/"\
> "FSC_0.01deg_201302210705_201302211215_MOD_panEU_ENVEOV2.1.00.tif"
$ gdalinfo "$IRREGULAR3D_SAMPLE" | grep 'Band'
  Band 1 Block=5600x1 Type=Byte, ColorInterp=Gray
$ gdalinfo "$IRREGULAR3D_SAMPLE" | grep '^Size'
  Size is 5600, 3700
$ gdalinfo "$IRREGULAR3D_SAMPLE" | grep 'PROJCS'
$ gdalinfo "$IRREGULAR3D_SAMPLE" | grep 'GEOGCS'
  GEOGCS["WGS 84",
$ gdalinfo "$IRREGULAR3D_SAMPLE" | grep 'Origin' -A1
  Origin = (-10.999999999999996,72.000000000000000)
  Pixel Size = (0.010000000000000,-0.010000000000000)
```

Listing 23 – Metadata for the irregular 3D time-series sample dataset.

This time we cope with 5600×3700 grayscale maps of $1\,^\circ$ of resolution on both latitude and longitude

axes. The CRS is the commonly (and wrongly) used geographic CRS based on the WGS84 datum (→EPSG:4326): we assume that the equirectangular Plate Carrée projection is applied and that we are working on a plane (from ellipsoidal to cartesian coordinate system).

Now concerning time, we need to decide how to encode the temporal information, but first of all *when* is our time-series? By properly parsing the file names of each image we are going to ingest, we can extract the time information (Listing 24). We have a resolution which is down to the minutes, so it can be a good trade-off between simplicity and meaning of time coordinates to choose the well-known UNIX time indexing: linear count of seconds from January 1st 1970 (option of the bash `date` command).

```
$ for map in $( ls "$IRREGULAR3D" | awk -F '_' '{ print $4 }' ); do
> date -ud "${map:0:8} ${map:8:4}" +%F\T%R\ =\ %s\ Unix
> done
  2011-02-01T12:50 = 1296564600 Unix
  2011-02-02T12:00 = 1296648000 Unix
  2011-02-03T12:40 = 1296736800 Unix
  2011-02-04T11:50 = 1296820200 Unix
  2011-02-05T12:30 = 1296909000 Unix
  2013-02-20T12:55 = 1361364900 Unix
  2013-02-21T12:15 = 1361448900 Unix
  2013-02-22T12:55 = 1361537700 Unix
  2013-02-23T12:00 = 1361620800 Unix
  2013-02-24T12:40 = 1361709600 Unix
```

Listing 24 – Getting UNIX time of the images of *IrregularTimeSeries*.

Now that we know the numeric time coordinates associated with each image, we have sufficient information to proceed with the usual *rasgeo* ingestion. With respect to the ingestion of a regular cube presented in the previous section, we now have to take care of two more `rasimport` options:

`--z-coords`
: Here we set all the (colon-separated) absolute time coordinates associated with each image in the irregular time-series: here is where we will put our UNIX time instants.

`--crs-order`
: As many geographical CRSs, EPSG:4326 defines latitude as the first axis. Since we strictly refer to the CRS definitions for ordering the coordinates in the CRS tuples, we have to tell rasimport that in this case the first grid axis (always in the horizontal East/West direction) is not parallel to the latitude CRS axis, but rather to longitude, and we do this by assigning an *order* to each axis.

Having understood these two new `rasimport` arguments, then you can see how the ingestion will finally look like in Listing 25.

As you can appreciate from the RasQL `sdom` request, despite the irregular spacing in the external time CRs, our ten images got stacked together along the third marray dimension without any empty cells (`sdom[m](2) = 0:9`).

Looking more closely at the WCS new *IrregularTimeSeries* coverage description, we can see that several *coefficients* got attached to the offset vector associated to the temporal axis. The offset

```
$ # data type
$ rasdl -p | grep GreyCube
  typedef marray <char, 3> GreyCube;
  typedef set <GreyCube> GreySet3;
$ # rasimport help
$ rasimport | egrep '^ --crs-order|^ --z-coords'
  --crs-order  order in which CRSs are specified by the --crs-uri identifer(s)
  --z-coords   irregularly spaced z-axis coordinate(s) of the 2D image slice(s)
$ # ingest
$ rasimport -d "$IRREGULAR3D" -s 'tif' \
> --coll 'IrregularTimeS' \
> --coverage-name 'IrregularTimeSeries' \
> -t GreyCube:GreySet3 \
> --crs-uri '%SECORE_URL%/crs/EPSG/0/4326':\
> '%SECORE_URL%/crs/OGC/0/UnixTime'\
> --crs-order 1:0:2 \
> --csz 1 \
> --z-coords \
> 1296564600:1296648000:1296736800:1296820200:1296909000:\
> 1361364900:1361448900:1361537700:1361620800:1361709600
$ # check
$ rasql -q "select sdom(m) from IrregularTimeS as m" --out string | grep Result
  Result element 1: [0:5599,0:3699,0:9]
$ rasql -q "select m[5599,3699,6] from IrregularTimeS as m" --out string | \
> grep Result
  Result element 1: 30
$ python
>>> import Image, numpy, os
>>> numpy.array(Image.open(os.environ['IRREGULAR3D_SAMPLE']))[3699,5599]
    30
$ wget "${WCS2_ENDPOINT}?service=WCS&version=2.0.1&"\
> "request=DescribeCoverage&"\
> "coverageid=IrregularTimeSeries" -O describeSnow3D.xml
$ grep Corner describeSnow3D.xml
  <lowerCorner>35 -11 1296564600</lowerCorner>
  <upperCorner>72 45 1361709600</upperCorner>
$ grep -o 'axisLabels=.* ' describeSnow3D.xml | head -n 1
  axisLabels="Lat Long unix" uomLabels="degree degree s"
```

Listing 25 – Ingesting the irregular 3D time-series.

vector is equal to 1 (second, [s]), exactly as we told to *rasgeo* via the `--csz` argument, but the coefficients are not equal to the `--z-coords` that we had assigned.

The temporal metadata is however correct: we pass to *rasgeo* handy absolute time coordinates, that are then translated to *relative* coordinates, or better said coefficients. In order to retrieve the absolute time coordinate T_i of the i^{th} time slice:

$$T_i = T_{origin} + (v_T * c_i)$$

being T_{origin} the absolute time coordinate of the grid origin, v_T the chosen offset vector along time, and c_i the i^{th} (weighting) coefficient. For example, the absolute UNIX time coordinate of the third image of *IrregularTimeSeries* is given by: $1296564600 + (172200 * 1) = 1296736800$ [s]. This is furtherly explained in Listing 26.

Now that we have correctly imported our snow images into the database we are finally going to propose some WCS and WCPS query examples.

```
$ grep 'coefficients>' describeSnow3D.xml
  0 83400 172200 255600 344400 64800300 64884300 64973100 65056200 65145000
$ for map in $( ls "$IRREGULAR3D" | awk -F '_' '{ print $4 }' ); do
> echo $(( $( date -ud "${map:0:8} ${map:8:4}" +%s ) - 1296564600 ))
> done
  0
  83400
  172200
  255600
  344400
  64800300
  64884300
  64973100
  65056200
  65145000
```

Listing 26 – Coefficients associated to the offset vector along the irregular temporal dimension of the *IrregularTimeSeries* coverage.

2.3.2 WCS in action

In the first place, we will surprisingly try to get a WCS exception from our server: in Listing 28 we actually demonstrate that our coverage has a 0-dimensional footprint along time. We deliberately miss our time-slice of just 1 nanosecond, but still you will see how the server will complain about it.

```
${WCS2_ENDPOINT}?
  request=GetCoverage&
  coverageid=IrregularTimeSeries&
  subset=Lat(50,50.02)&
  subset=Long(-7,-6.98)&
  subset=unix("2013-02-24T12:40:00.001Z")
```

Listing 27 – Sample size on an irregular dimension is 0: even though being close 1 nanosecond to a time-slice, this request on *IrregularTimeSeries* will return a WCS exception.

Now let's see some data instead, and again we start with a WCS subsetting to extract a image from the irregular series, for instance the image at 12:40 PM on February 24th 2013. This time we also want to exploit the WCS *Scaling* extension to reduce the output size by a factor of 10 (using `scalefactor` key) and then we encode it as a GeoTIFF (see Listing 28). You can viualize the response to this *GetCoverage* request in Figure 2.5.

```
${WCS2_ENDPOINT}?
  request=GetCoverage&
  coverageid=IrregularTimeSeries&
  subset=unix("2013-02-24T12:40Z")&
  scalefactor=10&
  format=image/tiff
```

Listing 28 – Reducing size by a scaling factor of 10 on a time slice of the *IrregularTimeSeries* coverage via WCS. The response is shown in Figure 2.5.

Which interpolation has been applied to our reponse image? As declared in the service capabilities document (response to a WCS *GetCapabilities* request), the only interpolation method that our

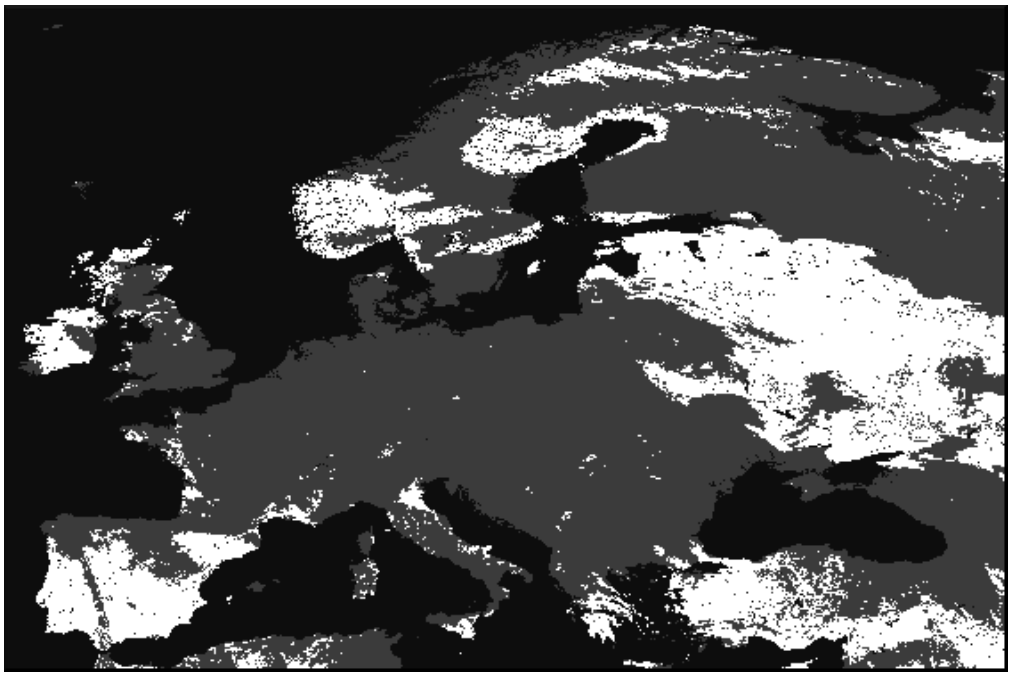

Figure 2.5 – Visual of the output of the WCS request from Listing. 28 : scaling down with nearest neighbor interpolation.

service can do now is nearest neighbor[6].

Last and not least example, we show now how to use the WCPS *coverage constructor* to trigger some simple server-side computations: for instance we want to count the exceedances of fractional snow cover with respect to a given threshold of, say, 30[7] over a region of interest and only for the year 2013.

As you can see in Listing 29, this is quite easily achieved by means of a simple WCPS query. The response is shown in Figure 2.6.

```
for cov in (IrregularTimeSeries)
return encode(
  coverage count_cov
  over $pxx x( imageCrsDomain(cov[Long(20:25)], Long) ),
       $pxy y( imageCrsDomain(cov[ Lat(40:42)],  Lat) )
  values count(
     cov[Long($pxx),Lat($pxy),unix("2013-01-01":*)] > 30
  ),
"csv")
```

Listing 29 – Counting the exceedances of fractional snow cover in *IrregularTimeSeries* from January 1st on, using the WCPS coverage constructor. A visual of the response is shown in Figure 2.6.

Again there are much more and much more advanced use-cases that can arise from using WCPS on these datasets (see the Cryoland portal), but this falls out of the scope of this tutorial, which has now reached the end.

[6]Note that scaling is only possible on regularly gridded axes: interpolation is done down at the marray level where irregular positions are unknown, inhibiting proper scaling of irregular dimensions.

[7]That is the binary value, which is mapped to some fraction by means of an external palette.

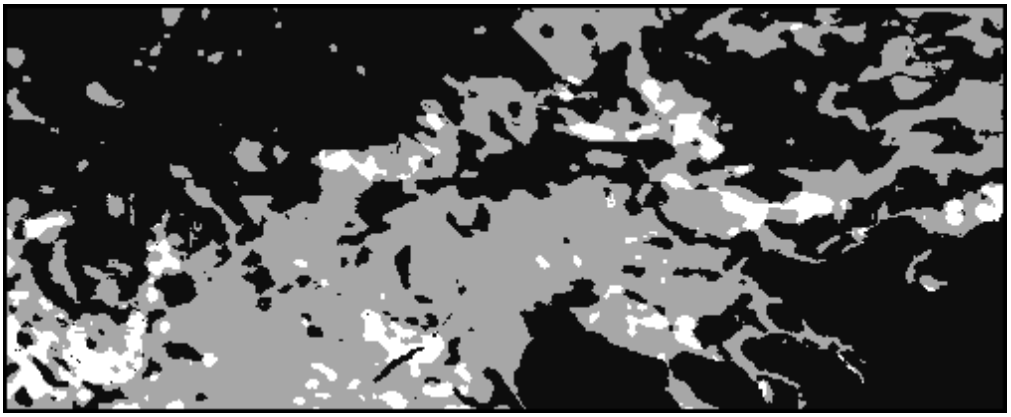

Figure 2.6 – Visual of the output of the WCPS request from Listing. 29 : counting the number of fractional snow cover exceedances over a region of interest.

For any question we are pleased to help you via our mailing lists for users (*rasdaman-users@googlegroups.com*) and developers (*rasdaman-dev@googlegroups.com*), or otherwise you can also dare to contact us directly (see title page).

The central hub of the `rasdaman` community is at `http://rasdaman.org`.

Bibliography

P. Baumann. OGC Web Coverage Processing Service (WCPS) Language Interface Standard. *OGC 08–068r2*, 2009.

P. Baumann. Beyond rasters: introducing the new OGC Web Coverage Service 2.0. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 320–329. ACM, 2010a.

P. Baumann. The OGC Web Coverage Processing Service (WCPS) standard. *Geoinformatica*, 14 (4):447–479, Oct. 2010b. ISSN 1384-6175. doi: 10.1007/s10707-009-0087-2. URL `http://dx.doi.org/10.1007/s10707-009-0087-2`.

P. Baumann. GML application schema for coverages. *OGC 09–146r2*, 2012.

P. Baumann. OGC Web Coverage Service (WCS) – Core. *OGC 09–110r4*, 2012.

P. Baumann. OGC Web Coverage Service 2.0 Interface Standard — KVP Protocol Binding Extension — Corrigendum. *OGC 09–147r3*, 2013.

P. Baumann and J. Yu. OGC Web Coverage Service Interface Standard — CRS Extension. *OGC 11–053*, 2014a.

P. Baumann and J. Yu. OGC Web Coverage Service Interface Standard — Interpolation Extension. *OGC 12–049*, 2014b.

P. Baumann and J. Yu. OGC Web Coverage Service WCS Interface Standard — Processing Extension. *OGC 08–059r4*, 2014.

P. Baumann and J. Yu. OGC Web Coverage Service Interface Standard — Range Subsetting Extension. *OGC 12–040*, 2014a.

P. Baumann and J. Yu. OGC Web Coverage Service Interface Standard — Scaling Extension. *OGC 12–039*, 2014b.

P. Baumann, P. Furtado, R. Ritsch, and N. Widmann. The rasdaman approach to multidimensional database management. In *Proceedings of the 1997 ACM symposium on Applied computing*, pages 166–173. ACM, 1997.

P. Baumann, P. Campalani, D. Misev, and J. Yu. Finding my CRS: A systematic way of identifying CRSs. In *ACM SIGSPATIAL GIS*, pages 71–78, November 2012.

P. Campalani, A. Beccati, and P. Baumann. Improving efficiency of grid representation in GML. In *27th International Conference on Informatics for Environmental Protection (EnviroInfo)*, pages 703–708. Shaker, Sep 2013a.

P. Campalani, D. Misev, A. Beccati, and P. Baumann. Making time just another axis in geospatial services. In *20th International Symposium on Temporal Representation and Reasoning (TIME)*, Sep 2013b.

P. Campalani, A. Beccati, S. Mantovani, and P. Baumann. Temporal analysis of atmospheric data using open standards. In *4th Symposium on Geospatial Databases And Location Based Services*. ISPRS Technical Commission, May 2014.

A. J. Hey, S. Tansley, K. M. Tolle, et al. The fourth paradigm: data-intensive scientific discovery. 2009.

D. Misev, M. Rusu, and P. Baumann. A semantic resolver for coordinate reference systems. *Web and Wireless Geographical Information Systems*, pages 47–56, 2012.

S. Natali, A. Beccati, S. D'Elia, M. Veratelli, P. Campalani, M. Folegani, and S. Mantovani. Multitemporal data management and exploitation infrastructure. In *The 6th International Workshop on the Analysis of Multi-temporal Remote Sensing Images (MultiTemp)*, Jul 2011.

OGC. The OpenGIS Abstract Specification — Topic 6: Schema for coverage geometry and functions. *OGC 07–011*, 2006.

J. Passmore. Opening up access to geological data with Rasdaman based WCS and WCPS services. Presentation at FOSS4G, 2013.

T. Perperis, P. Koltsida, and G. Kakaletris. Earthserver: Information retrieval and query language. In *EGU General Assembly Conference Abstracts*, volume 15, page 5740, 2013.

C. Portele. OGC Geography Markup Language (GML) Encoding Standard. *OGC 07–036*, 2007.

C. Portele. GML — Extended schemas and encoding rules. *OGC 10–129r1*, 2012.

rasdaman Gmbh. *rasdaman Query Language Guide*, 9.0 edition, 2014.

A. Robin. OGC SWE Common Data Model Encoding Standard. *OGC 08–094r1*, 2011.