28/08/2025
Version 1.0

BMD
Biodiversity Meets Data

M16 - First data cubes ready

Author(s): Niels Billiet
Contributor(s): Mathias Dillen

Co-funded by
the European Union

Project funded by

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Economic Affairs,
Education and Research EAER
State Secretariat for Education,
Research and Innovation SERI

BMD (Biodiversity Meets Data) receives funding from the European Union’s Horizon Europe Research and Innovation Programme and the Swiss State Secretariat for Education, Research and Innovation (SERI) (ID No 101181294). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union, the European Research Executive Agency (REA) or SERI. The EU, REA and SERI cannot be held responsible for them.

Prepared under contract from the European Commission
Grant agreement No. 101181294
EU Horizon Europe Research and Innovation Action

Project acronym:	**BMD**
Project full title:	**Biodiversity Meets Data**
Project duration:	01.03.2025 – 28.02.2029 (48 months)
Project coordinator:	Stichting Naturalis Biodiversity Center (Naturalis)
Call:	HORIZON-CL6-2024-BIODIV-01
Milestone title:	First data cubes ready
Milestone n°:	M16
Means of verification:	Report and software repository
Work package:	WP3
Nature of the milestone:	Document and software
Contribution to deliverable n°:	D3.1
Licence of use:	CC-BY
Lead beneficiary:	MeiseBG
Recommended citation:	Billiet,N. , Dillen, M. (2025). ***First data cubes ready***. BMD project deliverable M16.
Due date of milestone:	Month n° 6
Actual submission date:	28 Aug 2025
Quality review:	Yes

Milestone status:

Version	Status	Date	Author(s)	Actions
0.1	Draft	01 Aug 2025	Niels Billiet, Mathias Dillen (MeiseBG)	Sent for review
0.2	Draft	15 Aug 2025	Julian Oeser (UFZ), Sharif Islam (Naturalis)	Reviewed
0.3	Draft	27 Aug 2025	Niels Billiet, Mathias Dillen (MeiseBG)	Finalised, with incorporation of feedback from reviewers
1.0	Final	28 Aug 2025		Submitted

Table of contents

Summary

In this milestone our goal is to provide a prototype for the cubing engine (deliverable D3.1), that can serve as a starting point for a more directed software development process with feedback from developers of other elements of the BMD infrastructure in other WPs, in particular the catalogue in WP2, the data space in WP4 and the VREs in WP5. The prototype serves in particular as a demonstration of how the bridge between the data catalogue and the effective use of the data can be facilitated.

For the development of the cubing engine prototype we have focused on tools that facilitate the generation of data products that can be used in a research environment that seeks to implement Species Distribution Models (SDM), a common problem in the field of biodiversity research. To this end we will be integrating two data sources which have already been catalogued by WP2, namely GBIF (as a source of biodiversity data) and CHELSA (as a source of abiotic climate data). In addition to limiting the data sources to these two, we will also choose to develop and test our prototype with the pilot test case of a Belgian Natura 2000 site, namely the Soniënforest and the surrounding areas.

The cubing engine and the associated functionalities will be implemented in *Python 3* and will use the *xarray* library as its basis. *Xarray* has been chosen because it has been developed with geospatial modeling in mind and the implemented data structures naturally lean towards the concept of datacubes and supports various packaging formats. For the cubing engine itself we have decided to go with a hierarchical class scheme, a modular approach that allows for the mixing and matching of different data layer configurations based on the data catalogue. The end user interfaces with the engine itself by specifying their data requirements within a *yaml* file for which a template has been provided in the repository itself. This circumvents the need for the user to understand all the individual components that are present in the engine. The yaml file will be read by the different library objects and extract the relevant parameters that will be used to execute the data gathering and cubing.

For the data structure of the cube itself, the chosen data layers present an ideal challenge that informs how the end product of the data cube will look like. Due to the nature of the data we can not construct one overarching cube that combines all the different layers together. This is mainly due to the presence of unique dimensions in some of the layers, which would cause memory bloat in the end product as all datasets must share the same dimensions. Missing dimensions will be cast in the datasets and contain invalid entries, causing unnecessary memory usage. As such we will provide the user with the option of exporting the data as a Data Tree object. This is an object that links individual datasets together within a tree structure without forcing them to be represented within the exact same dimensions. Relationships between the data cubes are maintained however through the structuring of the tree itself. Beside this added benefit it also allows us to combine raster data with vector data. The conversion from vector data to raster data can be quite intensive and as such we choose to keep the data in its current form. Further consultation with stakeholders and users is required to determine the needs of the data end product and the required functionality.

List of abbreviations

EU	European Union
GBIF	Global Biodiversity Information Facility
CHELSA	Climatologies at High resolution for the Earth's Land Surface Areas
SDM	Species Distribution Modeling
VRE	Virtual Research Environment
OLAP	OnLine Analytical Processing
OLTP	OnLine Transaction Processing
CRS	Coordinate Reference System
EEA	European Environment Agency
CMIP	Coupled Model Intercomparison Project
API	Application Programming Interface
EEA	European Environment Agency
SQL	Structured Query Language
bbox	Bounding Box
wkt	Well Known Text representation

1. Introduction

In this section we will give a short outline of

- What a data cube is
- What a data cube means in the context of geospatial modeling
- What data harmonization is
- What data harmonization means in the context of geospatial modeling

1.1. Data Cubes

1.1.1. Introduction to Data Cubes

In information and data science a data cube can generally be defined as a multidimensional (n-D) data array that is utilized in organizing and grouping different sources of data together in an overarching structure. This requires that the data that is being grouped together into this structure describe variables that are defined along the same dimensions with the same coordinates.

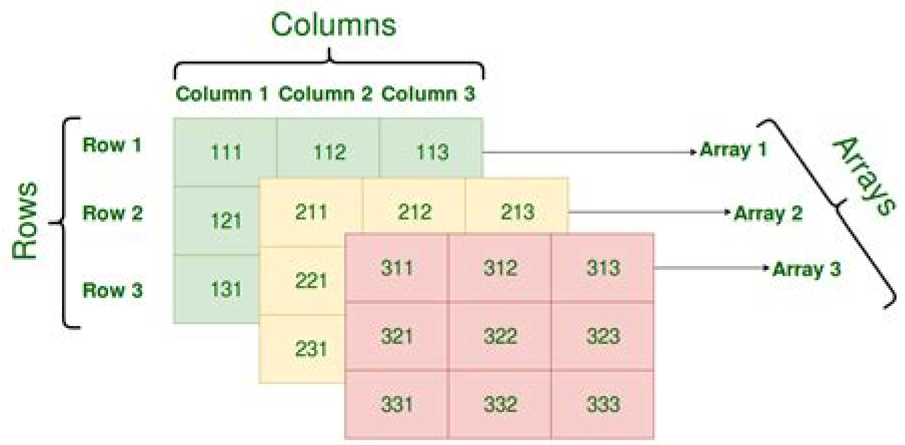


Figure 1: A visual representation of a 3D data cube. Cubes can be seen as the generalization of the concept of arrays to data where each array share columns and rows and can be combined together

Fig. 1 demonstrates the simplest representation of a data cube where we have a collection of data that can be described along 3 dimensions. Each array in this example represents a different variable that contains its own unique set of values. The arrays thus differ in what values they can take on but they share the row and column dimensions. This allows us to group arrays together and form an overarching data structure in which the structure itself informs us that its entries are compatible and can be overlaid with each other.

A data cube is not limited to 3 dimensions however, and can be extended to N dimensions depending on the data which is being cubed. The only condition that is imposed is that for an N dimensional cube, (N-1) of its dimensions are shared and 1 dimension is unique, this one dimension being the variables and their associated values.

In data science and specifically database structure we make a general distinction between OLAP and OLTP data sources. An OLAP or OnLine Analytical Processing structure, as its name suggests, is optimized for analytical purposes meaning it is well suited for subsetting, transformations, etc. Cubes are inherently an OLAP data structure because the data is described through a set of dimensions that are described by predefined coordinates. Structuring the data along this format allows us to perform the following actions efficiently (see Fig.2).

- Slicing, where we subset the data for a static value for a given variable. E.g. we can take out a temporal slice by only considering data for a given year and month combination.
- Dicing, where we subset the data in a given range for a set of variables. E.g. on top of this temporal slice we limit the data within a specified spatial extent like a small bounding box within a large bounding box
- Drill-down, where we increase the resolution for a set of variables. E.g. this operation can be interpreted multiple ways. Suppose we have data that is continuous at a specified resolution but want to go to a finer resolution than we can resample the data and interpolate. For discrete classes that follow a hierarchy we can go further down this hierarchy to more specific classes. A natural example here would be taxonomy ranks where we go from a higher to a lower rank, e.g. from class to species level.
- Roll-up, where we decrease the resolution for a set of variables. This operation is the inverse of the drill-down operation.
- Pivoting, where we change the ordering of the dimensions. E.g. suppose we have a dataset that is described according to (latitude, longitude, time) than we can easily change this to (longitude, latitude, time), (time, latitude, longitude), (time, longitude, latitude), (latitude, time, longitude) or (longitude, time, latitude).

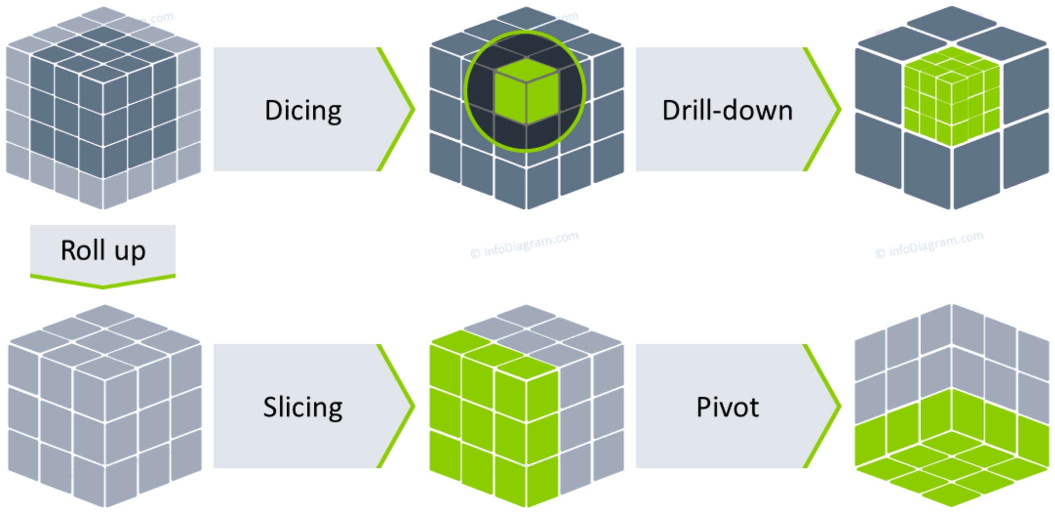


Figure 2: A visual representation of the different cube operations

OLAP data structures are contrasted with OLTP, or OnLine Transaction Processing data structures. These data structures are optimized for processing large amounts of data transactions such as adding records, removing records or modifying records. An example of this are SQL data tables where each record is indexed so that insertion, removal or modification is easy. These systems are inherently suboptimal for the operations that are characteristic for OLAP data structures due to the fact that we have to iterate over each column individually for filtering operations. This is circumvented in data cubes due to the ordering along fixed and ordered dimensions.

1.1.2.Geospatial Data Cubes

Within the context of geospatial modeling, the concept of a data cube is something that is inherently compatible with the nature of the data being employed within it on multiple levels. For geospatial data, all the data is described within spatial dimensions and optionally a temporal dimension. Moreover, much geospatial data can be stored and processed as rasterized data, making it closely related to the structure of a data cube.

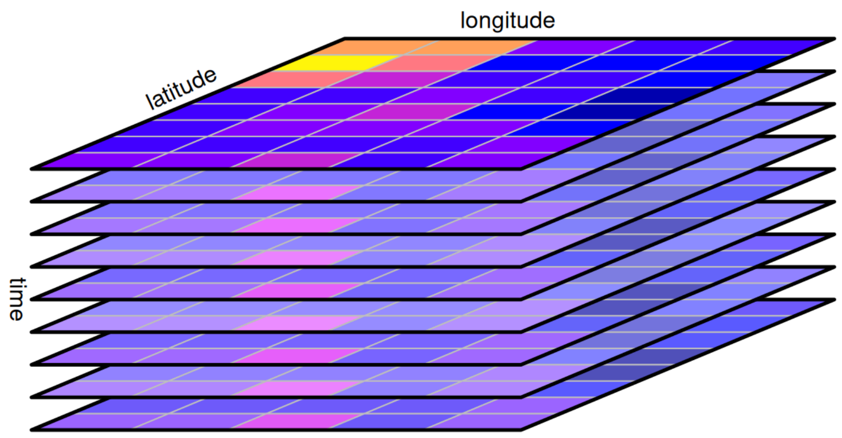


Figure 3: Example visualization of a geospatial raster datacube for a single variable with spatial dimensions (longitude, latitude) and a temporal dimension

Rasters are not the only way of representing data in geospatial science. A common alternative is so-called vector data, however this is no issue as vector data can also be represented within a cube format. The main difference between raster data and vector data is that raster data represents data as a rectangular grid consisting of (typically) square cells, whereas vector data associates data with 2D geometrical elements such as points, lines and polygons. Vector data can be cubed as well by replacing the geospatial coordinate dimensions with a 'geometry' dimension where the geometrical elements of interest are the coordinates of the dimension. Fig.4 represents an example of this kind of data structure.

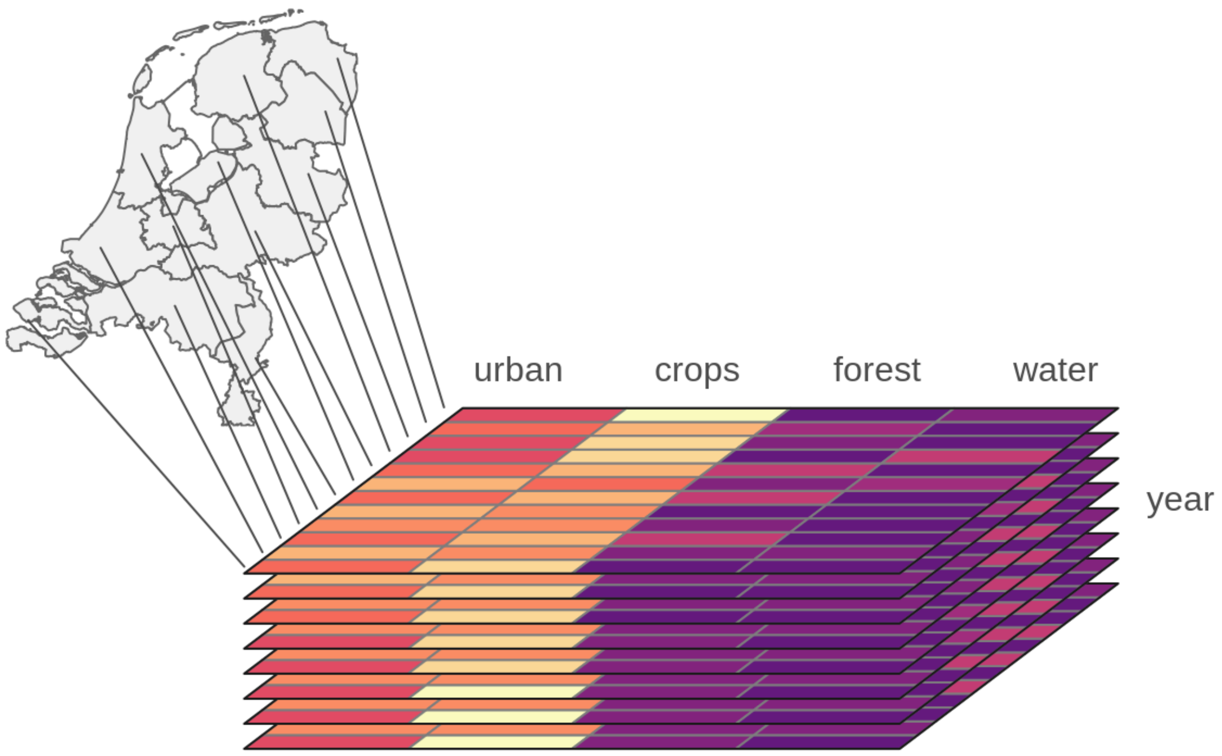


Figure 4: Example visualization of a geospatial vector datacube where one of the dimensions of the cube are the polygons that describe the borders of different administrative regions and the data is described for the region enclosed by this polygon.

1.2. Data Harmonization in geospatial cubes

As was mentioned in the introduction section of data cubes, the whole concept of data being cubeable depends on the assumption that the underlying data has shared dimensions which can be overlaid on top of each other to create a stacked data structure. However, requiring the data to share dimensions is not the only prerequisite for this process. In addition to shared dimensions, we also require the coordinates of these dimensions to:

- Share the same resolution, i.e. the spacing between coordinates (the size of the grid cells) is shared
- Have dimensions which are described within the same units, e.g. uniform units of distance and same formatting of time
- Use methods/measurements that employ consistent or similar references, like the zero point and the Coordinate Reference System (CRS)

Besides the harmonization of the data itself we can also take into account that harmonization can and should be performed in terms of metadata, consistent file types, data types, etc.

Within the context of harmonization of spatiotemporal data, this can entail multiple things:

- Data in geospatial modeling is referenced in geographical space based on Coordinate Reference Systems (CRS), which differ in the way spatial coordinates are described
- Rasters extents are not necessarily aligned, meaning the different data arrays are not overlapping
- Different variables can be recorded at different spatial and/or temporal resolutions

2. Prototype testing data and data layers

2.1. Prototype test problem

For the prototype, we will focus on one of the Natura2000 sites in Belgium: **the Sonian Forest and its surrounding areas**.

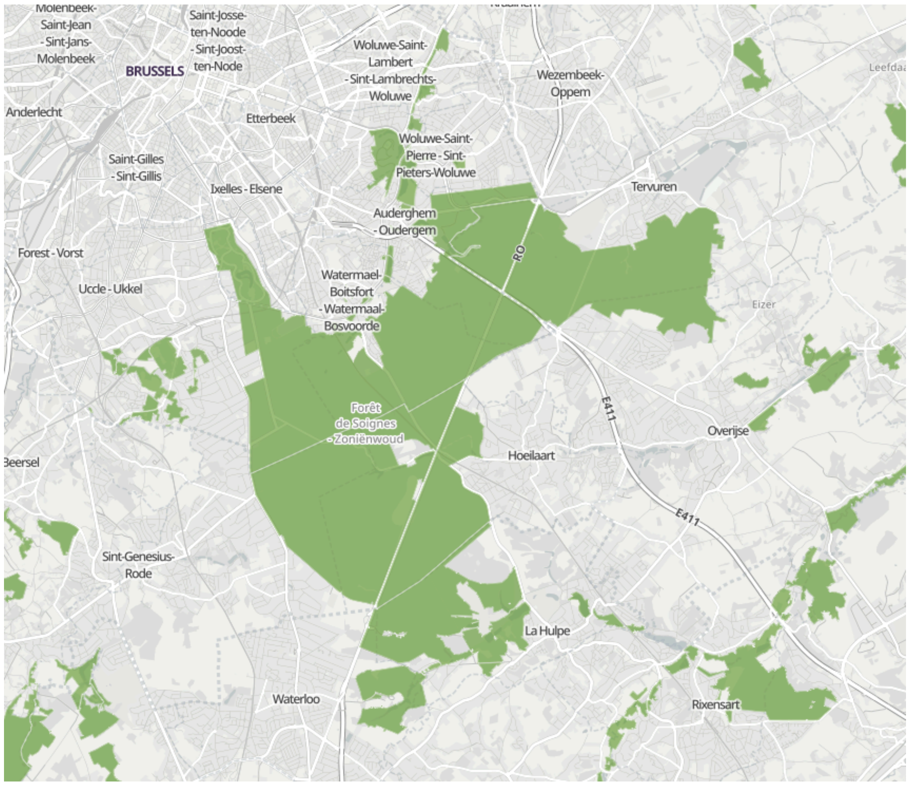


Figure 5: Sonian Forest and its surrounding Natura2000 areas

The Sonian Forest and its surroundings are characterized by a set of habitats, which are described on the Natura2000 website.

The areas of interest include:

1. **La Forêt de Soignes avec lisières et domaines boisés avoisinants et la Vallée de la Woluwe**
   Complexe Forêt de Soignes - Vallée de la Woluwe

- **Area code:** BE1000001
- **Protected under:** the Habitats Directive
- **Area:** 2066 ha
- **Protected:** 5 species & 8 habitats

2. **Sonian Forest**
    - **Area code:** BE2400008
    - **Protected under:** the Habitats Directive
    - **Area:** 2066 ha
    - **Protected:** 3 species & 9 habitats

3. **Vallées de l'Argentine et de la Lasne**
    - **Area code:** BE31002C0
    - **Protected under:** both Birds and Habitats Directives
    - **Area:** 821.45 ha
    - **Protected:** 16 species & 14 habitats

Each habitat is characterized by a set of species that indicate its ecological value. In total, there are **211 species of interest** in this area. These are listed in the file prototypeNames.csv, which can be found in the prototype script directory under the inp folder of the code repository. Additionally, we include a file listing invasive species known in Belgium. This list, the *Global Register of Introduced and Invasive Species - Belgium*, is available via GBIF as a Darwin Core (DwC) archive dataset.

Spatially, we will define the area by a bounding box that has the following coordinates in longitude and latitude

- Minimal decimal longitude of 4.171371 and a maximal longitude of 4.742004
- Minimal decimal latitude of 50.684060 and a maximal latitude of 50.877911

2.2.Data layers

2.2.1.GBIF

GBIF aggregates species occurrence data at a global level from different sources, ranging from scientific research and heritage institutions to citizen science initiatives like iNaturalist. The data aggregated this way is harmonized and validated to significant extent, through numerous quality tests, the enforcement of domain data standards like Darwin Core and the ongoing efforts to align with a taxonomic backbone (cf. Task 3.2 for more information on this). As such, the data that is offered through GBIF APIs offers essential biodiversity data which forms an integral part of the BMD project.

GBIF offers multiple APIs through which users can consult data, the APIs that will be used within the prototype and the final version of the cubing engine are the SQL API for data acquisition, the downloads API to retrieve metadata and the pygbif package as a wrapper to facilitate API access. Using these APIs we will query occurrence data based on the parameters that are characteristic to the prototype. When querying data from GBIF we will use the following selection and aggregation criteria:

- For the temporal dimension, we will request year and month values. Because CHELSA data runs from 1980 to 2020 we will query data in this year range.
- We will perform aggregation based on the EEA grid, which is an equal area vector polygon grid supported by GBIF's SQL API. Occurrences will be linked to a specific cell code. The spatial dimension will be described in terms of geometries
    - Records that do not have an uncertainty associated with them, i.e. NULL or NaN, we associate a coordinate of 1 km as the standard uncertainty to it.
- We will request all taxonomic identifiers for the different taxon ranks of the species of interest. As such we can describe occurrences on all different levels of taxonomic hierarchy
- The following filtering criteria will be included within the query
    - We will require the data to be characterized by spatial coordinates
    - We will require the data to be characterized by temporal coordinates
    - We will require the data to represent an occurrence record. GBIF allows us to select data based on an occurrenceStatus which can either be present or absent. For the prototype we limit ourselves to species presence. Other variables like abundance can also later be inferred from Darwin Core properties such as dwc:organismQuantity or dwc:individualCount, both available through the SQL API.
    - Occurrence between a requested time range will be queried
    - The occurrence records describe one of the species of interest to the prototype area

```
SELECT
        "year",
        "month",
        GBIF_EEARGCode(1000, decimalLatitude, decimalLongitude,
COALESCE(coordinateUncertaintyInMeters, 1000)) AS eeaCellCode,
        speciesKey,
        species,
        genusKey,
        genus,
        familyKey,
        family,
        orderKey,
        "order",
        classKey,
        class,
        COUNT(*) AS occurrences,
        COUNT(DISTINCT recordedBy) AS distinctObservers
    FROM
        occurrence
    WHERE
```

```
        GBIF_Within('{wkt}', decimalLatitude, decimalLongitude) = TRUE AND
        hasCoordinate = TRUE AND
        occurrenceStatus = 'PRESENT' AND
        NOT ARRAY_CONTAINS(issue, 'ZERO_COORDINATE') AND
        NOT ARRAY_CONTAINS(issue, 'COORDINATE_OUT_OF_RANGE') AND
        NOT ARRAY_CONTAINS(issue, 'COORDINATE_INVALID') AND
        NOT ARRAY_CONTAINS(issue, 'COUNTRY_COORDINATE_MISMATCH') AND
        "year" IS NOT NULL AND
        "month" IS NOT NULL AND
        "year" >= {begin_year} AND "year" <= {end_year} AND
        taxonKey IN ({keys})
    GROUP BY
        species,
        speciesKey,
        eeaCellCode,
        "year",
        "month",
        genusKey,
        genus,
        familyKey,
        family,
        orderKey,
        "order",
        classKey,
        class
    ORDER BY
        "year" ASC,
        "month" ASC,
        speciesKey ASC
```

2.2.2.CHELSA

CHELSA is an initiative by the Swiss Federal Institute for Forest, Snow and Landscape research (WSL) that provides access to high spatial resolution (1 km) raster data yielding insights into current climate and simulated future climate scenarios. The data itself is made publicly available through their website which links to the metadata and the S3 storage bucket where the data is stored in GeoTIFF files, which are accessible for anyone.

In CHELSA, we make a distinction between different data layers that are of interest based on the temporal range covered.

- CHELSA time series which provide insights on a monthly basis between 1979 and 2020
- CHELSA climatologies series provide climatological data that spans 30 year intervals for both past and future data. The variables that are available here are all the BIOCLIM+ variables
    - Reference climatological data is being provided between 1981 and 2010.
    - Future simulation data from both CMIP5 and CMIP6, with different model and ensemble combinations for a wide range of future predictions. The future simulations can span 3 different year ranges namely 2011-2040, 2041-2070 and 2071-2100.

For the BMD project, all these data layers are relevant as they are frequently used in the context of SDMs and as such functionality for all these individual layers will be provided. Due to the lack of a provided API by the organization itself, custom functionality will be implemented within this prototype.

3. Software Architecture

The code that was written can be found in the BmC Github repository.

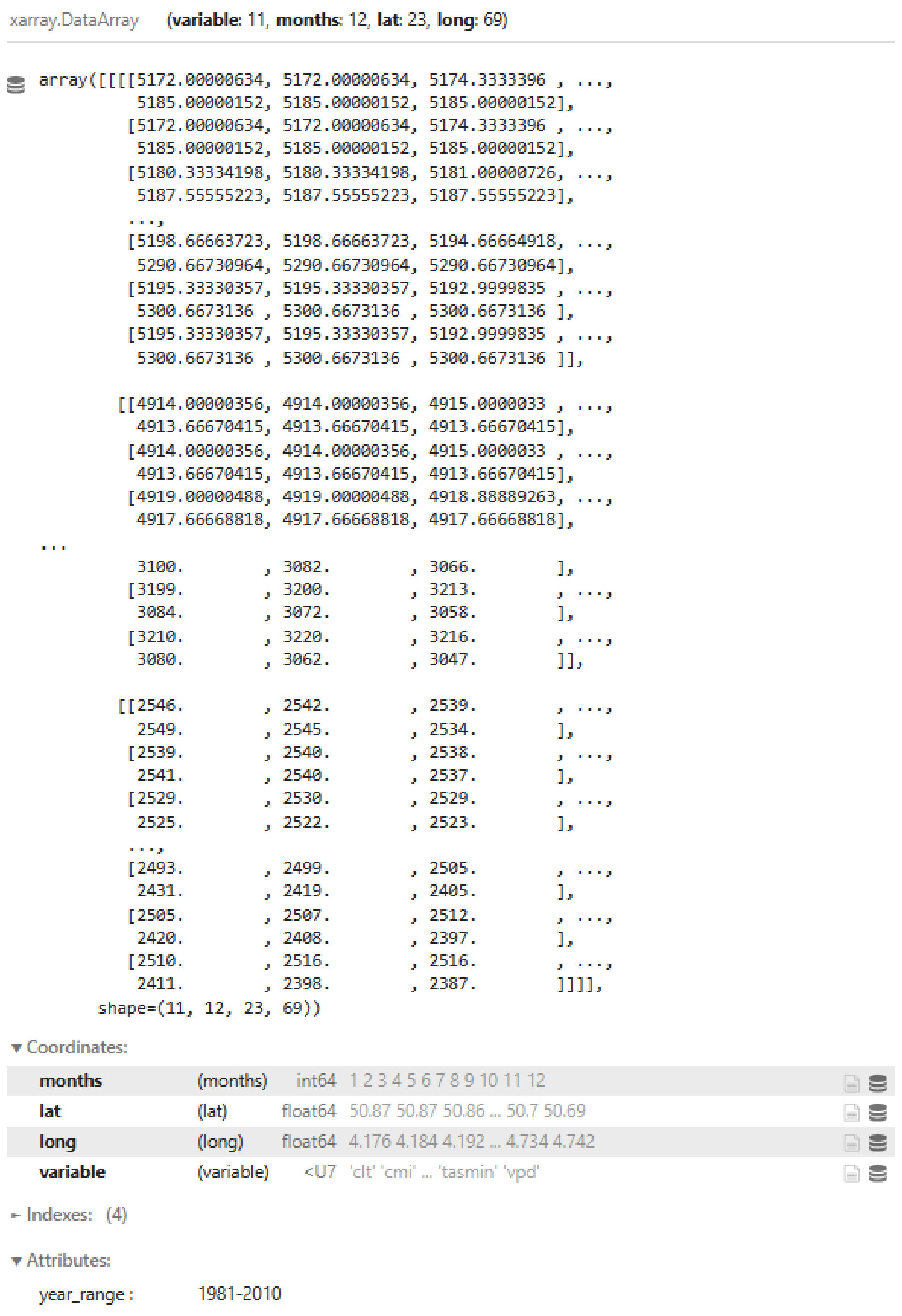
3.1. Xarray

Xarray is a python package that was specifically designed to handle geospatial data and has been built to interface smoothly with the data processing packages *numpy*, *pandas* and *dask*. Due to this foundation the library has been optimized for array manipulation, efficient querying and handling of large volumes of data in distributed systems.

Fundamental to xarray is a hierarchy of data structures built on top of each other, which naturally extends to a data cube concept. The foundational data structure is the data array which generalizes the concept of the numpy array. The main differences between a numpy array and xarray data array are:

- Xarray data arrays are constructed using 3 mandatory components
    - Data component, which contains the values of the variable. These values are presented as a multidimensional numpy array
    - Dimensions component, which defines the grid in which the data component is described. If we refer to the example of the geospatial data cube example as was illustrated in Fig.3, this would entail longitude, latitude and time.
    - Coordinates component, which provides the individual values that these dimensions can take. Suppose we have a time dimension in which we have data available between the years 2020 and 2025 than the coordinates for this dimension would be [2020, 2021, 2022, 2023, 2024, 2025]. The coordinates thus represent an ordered set of possible values that our data variables can be described in.
- In addition to these mandatory components we can add attributes to the arrays themselves which allows us to build metadata within the data structure itself.

An example of this data structure can be seen in Fig.6 where each component is visually represented. Multiple data arrays can be combined in an overarching data structure called a dataset, assuming that the individual data arrays share dimensions. If data arrays contain additional dimensions in comparison to other data arrays present, then the dataset will cast the missing dimension to the other data arrays filling them with NaN values. It is evident that the data structure of the dataset is inherently linked to the concept of data cubes due to this enforcement of shared dimensions.



```
xarray.DataArray    (variable: 11, months: 12, lat: 23, long: 69)

array([[[[5172.00000634, 5172.00000634, 5174.3333396 , ...,
          5185.00000152, 5185.00000152, 5185.00000152],
         [5172.00000634, 5172.00000634, 5174.3333396 , ...,
          5185.00000152, 5185.00000152, 5185.00000152],
         [5180.33334198, 5180.33334198, 5181.00000726, ...,
          5187.55555223, 5187.55555223, 5187.55555223],
         ...,
         [5198.66663723, 5198.66663723, 5194.66664918, ...,
          5290.66730964, 5290.66730964, 5290.66730964],
         [5195.33330357, 5195.33330357, 5192.9999835 , ...,
          5300.6673136 , 5300.6673136 , 5300.6673136 ],
         [5195.33330357, 5195.33330357, 5192.9999835 , ...,
          5300.6673136 , 5300.6673136 , 5300.6673136 ]],

        [[4914.00000356, 4914.00000356, 4915.0000033 , ...,
          4913.66670415, 4913.66670415, 4913.66670415],
         [4914.00000356, 4914.00000356, 4915.0000033 , ...,
          4913.66670415, 4913.66670415, 4913.66670415],
         [4919.00000488, 4919.00000488, 4918.88889263, ...,
          4917.66668818, 4917.66668818, 4917.66668818],
...
          3100.        , 3082.        , 3066.        ],
         [3199.        , 3200.        , 3213.        , ...,
          3084.        , 3072.        , 3058.        ],
         [3210.        , 3220.        , 3216.        , ...,
          3080.        , 3062.        , 3047.        ]],

        [[2546.        , 2542.        , 2539.        , ...,
          2549.        , 2545.        , 2534.        ],
         [2539.        , 2540.        , 2538.        , ...,
          2541.        , 2540.        , 2537.        ],
         [2529.        , 2530.        , 2529.        , ...,
          2525.        , 2522.        , 2523.        ],
         ...,
         [2493.        , 2499.        , 2505.        , ...,
          2431.        , 2419.        , 2405.        ],
         [2505.        , 2507.        , 2512.        , ...,
          2420.        , 2408.        , 2397.        ],
         [2510.        , 2516.        , 2516.        , ...,
          2411.        , 2398.        , 2387.        ]]]],
      shape=(11, 12, 23, 69))

▾ Coordinates:
  months     (months)    int64    1 2 3 4 5 6 7 8 9 10 11 12
  lat        (lat)       float64  50.87 50.87 50.86 ... 50.7 50.69
  long       (long)      float64  4.176 4.184 4.192 ... 4.734 4.742
  variable   (variable)  <U7      'clt' 'cmi' ... 'tasmin' 'vpd'

▸ Indexes:  (4)

▾ Attributes:
  year_range :   1981-2010
```

Figure 6: Example of a xarray.DataArray object

Due to the structure of the data array and having a foundation in pandas, the data array is also optimized for the different operations inherent to a data cube. By explicitly indexing value through named dimensions with specified coordinates we can efficiently subset, aggregate and slice through the data stored in the array itself.

Data arrays describe geospatial information for one specific variable of interest. However often we wish to combine multiple variables for a given area of interest. This is where the *xarray* Dataset object comes into play. The Dataset class can be seen as the overcoupling data structure that seeks to combine different data arrays in a unified object. Important to note here is that the Dataset class will assume that each DataArray is described along the same dimensions and coordinates. If this is not the case, missing dimensions will be added and coordinates will be reshifted without filling in the empty space with meaningful information.

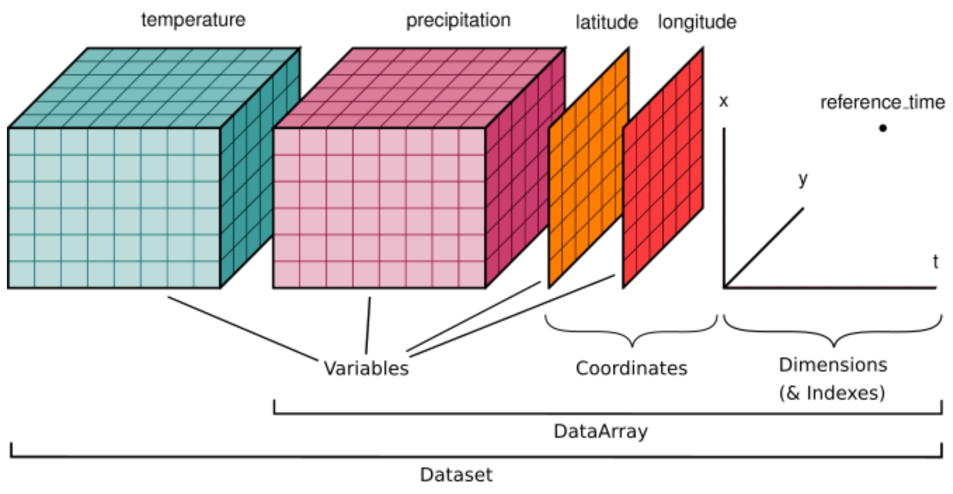


Figure 7: Overview of the data structures in xarray

In case we have different data arrays and/or datasets that do not necessarily share all their dimensions with each other but are still closely related, i.e. they might describe the same area of interests, we can utilize a Data Tree data structure. The data tree can be regarded as a hierarchical ordering of individual datasets in xarray, this ordering is imposed by the structure of the tree itself.

A tree (e.g. Fig.8) can be considered to be a data structure in which some data, also referred to as a node, can connect to other pieces of data and where this relationship can be seen as going into more depth. Essential to a tree is that this occurs in a specific hierarchy where we go from the top node, also referred to as a root, to lower roots, also referred to as branches, and eventually end up in the lowest level, which are called leaves. The leaves can be considered to be properties of the branches which are properties of the root. This concept can be applied in the cubing context as well where several cubes are describing the same root. However due to incompatibility in their dimensions we can't aggregate them into a single cube. Here the data tree offers a solution where the individual cubes are still aggregated together but due to the structure within the tree.

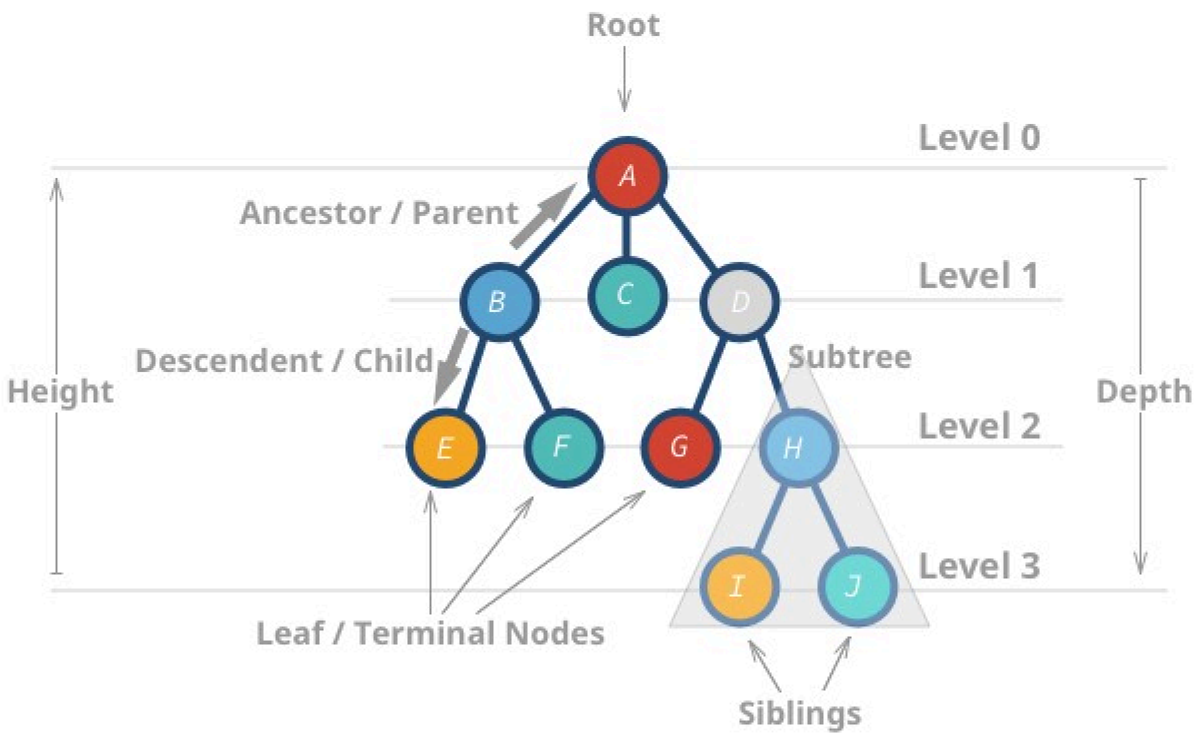


Figure 8: General schema of a tree data structure.

3.2. Datasource submodule

3.2.1.CHELSA

3.2.1.1. S3

All the CHELSA data is available in a publicly accessible S3 bucket where the CHELSA files are stored according to the time frame that they describe. The different files follow a standard naming scheme

- CHELSA_[short_name]_[timeperiod]_[Version].tif, for the monthly timeseries
- CHELSA_[short_name]_[timeperiod]_[model] _[ssp] _[Version].tif, for the different climatological data layers

In order to fetch the data based on the parameters specified by the user, we require URLs that link to the individual files that are stored in the S3 bucket. To do so we have written 5 functions that generate all possible URLs stored in the bucket:

- Format_url_month_ts, to generate URLs for the monthly time series
- Format_url_clim_ref_period, to generate the URLs for the reference historical climatological data containing aggregates for the entire period
- Format_url_clim_ref_month, to generate the URLs for the reference historical climatological data for period containing aggregate for month over the entire period

- Format_url_clim_sim_period, to generate the URLs for the simulated future climatological data which contains aggregates over the available periods in a given model and scenario
- Format_url_clim_sim_month, to generate the URLs for the simulated future climatological data which contains aggregates for each month over the available periods in a given model and ensemble

Although the metadata catalogued in the technical specifications lists the names of the file following a standard naming scheme for the different files, anomalies have been detected for several files which deviate from this proposed scheme. These anomalies have been hard coded into the function and have been shared with the developers of the data catalogue in WP2. The following issues have been documented:

- In the monthly time series the following deviations have been documented:
    - The variable `rsds` switches the position of month and year in the standard naming scheme
    - The variable `pet` adds `_penman` to the URL, whereas other files solely use the variable name which is also the directory name
- In the reference climatological data for period statistics the following deviation has been documented:
    - The variable `rsds` has multiple statistics associated with it such as `rsds_{stat}`, where stat is either range, max, min or mean. In all other variables who have these derived statistics associated with them, they stay together in the `[short_name]` section of the naming scheme. For the `rsds` variable however they get split and pasted after the time period.
- In the reference climatological data for month statistics in the reference period the same deviations as the monthly time series have been recorded, i.e. `rsds` and `pet`.
- No issues to date with the simulated data URLs

3.2.1.2. Sampling

The sampling functions are written to interface with the tiff files stored in the S3 bucket. These TIFF files are cloud-optimized GeoTIFFs, which enables users to request subsets of the relatively large files and not download them whole. We allow 2 modes of subsetting, namely

- *Read_bounding_box*, a function that extracts a subset of the cloud hosted tif file based on a window defined by a pair of longitude and latitude coordinate pairs. These coordinate pairs are the coordinates of the lower left and upper right corners of the window
- *Read_polygon_area*, a function that takes in a geometry stored within a shapefile which is subsequently used to subset all the data contained within its boundaries.

Both of these functions return the subsetted data as a numpy array and optionally (but standardly) also return the longitude and latitude coordinates in which the subsetted data is described. These generated coordinates are used in the `layer` functions subsequently.

3.2.1.3. Layer

The layer submodule contains all the functions that perform the data ingestion and structuring into a consistent object, namely the data array class from xarray. The following functions are provided in this library

- *Generate_month_year_range*, used in the generation of monthly time series URLs. This function generates all (month,year) pairs in a given temporal range defined by a starting year, month and end year, month.
- *Batch_process_urls*, reads all URLs and retrieves the subset based on the specified bounding box while simultaneously writing a progress bar to the output buffer that allows the user to monitor the ingestion process.
- *Check_spatial_homo*, check spatial homogeneity across all the data that is being extracted in the batch processing function. Transformation to a data array only happens if spatial homogeneity is satisfied.
- Layer generating functions, that generate the data array for each data layer within CHELSA
  - *Chelsa_month_ts*
  - *Chelsa_clim_ref_period*
  - *Chelsa_clim_ref_month*
  - *Chelsa_clim_ref_period*
  - *Chelsa_clim_ref_month*

3.2.2.GBIF

3.2.2.1. GBIF_SQL

This submodule groups all the functions that are used within the construction of the GBIF SQL query. For the construction of the GBIF SQL query the library provides functionality to

- Read_species_names, a function that reads the names of the different species that are stored in a csv file and returns a list of all the unique names
- Fetch_taxon_info, a function that takes in a file containing species names, subsequently extracts all the names and matches the names to the GBIF taxonomic backbone. During matching any matches that result in a match of type `higherrank` or `none` are written to the mismatch_df. The user gets a species_df containing the good matches and a mismatch_df containing the dubious matches. In the species_df a column called acceptedUsageKey is provided that stores the unique identifier for the correct name of the species, i.e. synonyms are corrected
- `Extract_keys_dwc`, a function written to extract the species identifiers from a dwc archive.
- `Bbox2polygon_wkt`, converts the coordinates of bounding box to a wkt style string
- `Generate_json_query`, generates a SQL query which is subsequently dumped into a json object. This json query can be submitted to the SQL API through the command line

Currently the GBIF SQL functionality is still somewhat limited. Further investigation into how we can generalize this is needed. We plan to develop functionality in collaboration with the VRE developers in WP5 to address their more specific data requirements.

3.2.2.2.Layer

The layer functionality is aimed at producing the data array from the GBIF data that is obtained through the SQL API. The data from GBIF is returned into a tabulated form and thus requires restructuring in order to be represented through an xarray data array.

Due to xarray being built on pandas the conversion of a pandas dataframe to an xarray data array can occur quite easily. We only require the data frame to be indexed along the columns that we wish to use as the dimensions of our data array. Performing the indexation will automatically generate the coordinates which span the dimension and as such no custom restructuring of the data is required.

However, upon experimenting with the GBIF data we have stumbled upon key issues which need to be addressed with the partners of the BMD project. Due to the many taxonomic dimensions that are characteristic to biodiversity data, which also can be very spread out, the standard data array structure explodes in terms of used memory. For example, in the prototype we have the following dimensions and number of levels that characterize them

- Time dimension, number of levels is the amount of temporal slices that are being asked. Within the prototype SQL script we have requested all data from 1980 to 2020 on a monthly basis. The amount of levels is therefore 488.
- EEA cellcode dimension, the geometry dimension by which the GBIF SQL API performs its cubing. The amount of levels in this dimension is 1047
- Taxonomically we have the following dimensions
    - Specieskey dimension, identifies an occurrence on the species level. Based on the data 140 levels are present in the data itself. The list however contained 200 species of interest.
    - Genuskey dimension, identifies an occurrence on the genus level. Based on the data 110 levels are present in the data itself.
    - Familykey dimension, identifies an occurrence on the family level. Based on the data 60 levels are present in the data itself.
    - Classkey dimension, identifies an occurrence on the class level. Based on the data 8 levels are present in the data itself.

The resulting data array would have the dimensionality of (488x1047x140x110x60x8) and results in a memory overload where the size of the data array took up 2 TiB of memory.

In addition to this enormous size of the data array we are also dealing with data that is characterised by a high degree of sparsity. It is a known problem within biodiversity research that the data is far and spread out. Indeed, upon inspecting the dimensions in Fig.7 we can see that both the spatial and temporal dimensions are characterized by a lower non zero count than the taxonomic dimensions. These dimensions thus contribute in a large amount to the sparsity in the data itself. Additionally we see a strong imbalance within the different taxonomic levels.

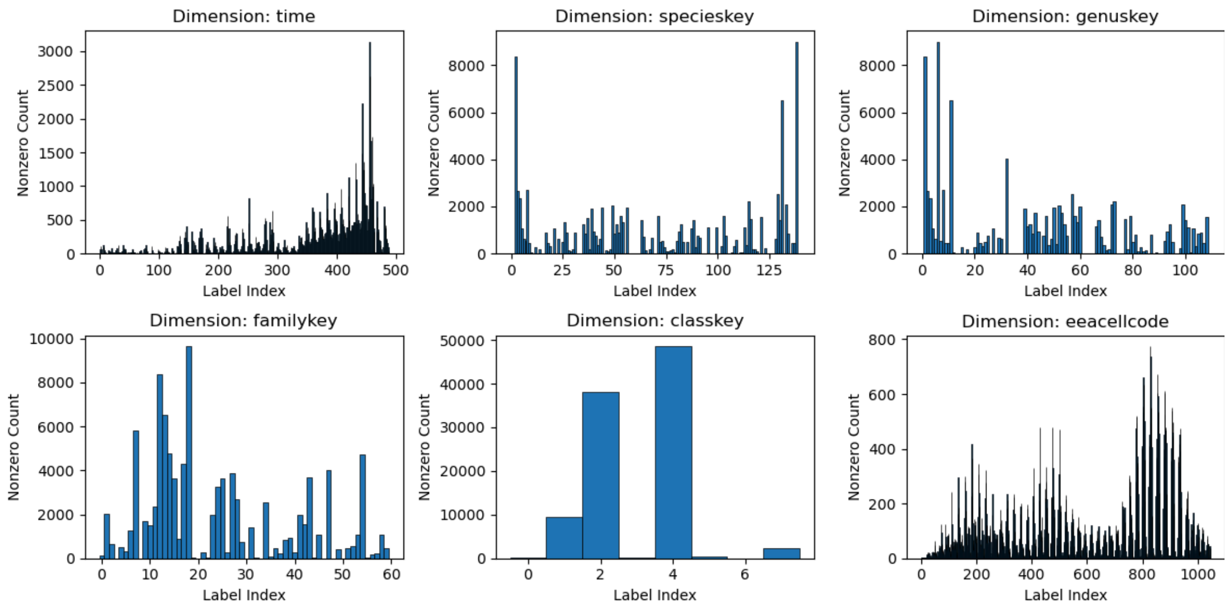


Figure 9: Distribution of the data density within the prototype GBIF data. For each dimension we plot the amount of non zero values recorded within the data. This allows us to visualize how the data is distributed within the dataset over the different levels.

As a solution to the high level of sparsity in the data we can represent the data as a sparse object where we store the individual data points as in terms of their coordinate indices instead of constructing the entire matrix. An example of this can be seen in Fig.10.

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Figure 10: An example of the coordinate index sparse representation method. For each value in the matrix we get the corresponding index for all the dimensions that they occupy within the matrix.

Using the sparse library in python and converting the data to a coordinate index representation we are able to reduce the size of the data array object from 2 TiB to 2MB. This roughly represents a compression with a factor of 10^6 highlighting the high degree of sparsity in the data.

However, despite the sparse representation solving the memory bloat that occurs with dense representation we still face issues with file formatting and exporting and importing this data in this representation. Netcdf is a file format that is built on the assumption that the user needs to export or import dense data and as such sparse data is not supported. Zarr supports some level of sparse data due to the fact that data is stored in a chunked manner which does allow some level of sparsity.

The current libraries do not inherently support the utilization of the sparse representation of data due to the fact that geospatial data in general always comes in a dense representation. Most environmental and climate variables are measured in a continuous fashion and thus produce dense data inherently. GBIF data on the other hand represent point observations where the domain of the taxonomic dimensions are large, resulting in data which is sparse in nature when the amount of data is not large enough.

A possible alternative in this situation could be found in TileDB, which integrates both sparse and dense data. GBIF and OBIS data are also available in (Geo)Parquet files in the form of occurrence snapshots. However, parquet files represent a tabular representation of the data which is not the desired data cube format that this cubing engine seeks to provide.

A temporary work around has been formulated which will be discussed in the bmd_cube class discussion in the following section.

3.3. Utils submodule

The utils submodule contains all functions that are used across the library and currently is only used for parameter reading functions. All parameters that need to be queried and processed into the cube are stored in a yaml configuration file which is explained more in depth in section 3.4.5. The util functions stored in extract_parm contain all functions necessary to deserialize the yaml file into a dictionary that can be passed to the cube classes.

3.4. Cube submodule

3.4.1. Class hierarchy and inheritance scheme

For the cubing engine itself we have chosen to make use of classes to define custom objects with controlled functionality. Fig.11 shows the inheritance scheme that was followed during the development of the cubing engine prototype.

- At the top of the scheme we find the `spatiotemporal_cube` from which all derived children classes draw fundamental geospatial functionality from. This ranges from functions that are
    - Involved in the construction and structuring of the data arrays for the different data layers
    - Involved in checking for the different conditions we require of the data arrays to satisfy. Specifically,we built the ability to check spatial and temporal coherence and consistency into the cube foundations itself

    - Involved in common grid operations which might/will be necessary during processing of the different data arrays. Among these functions we can find grid realignment and grid resampling
- The middle of the scheme involves the data layer cubes which are directly descended from the `spatiotemporal_cube`. In the current prototype engine we will provide the `CHELSA_cube` and the `GBIF_cube` as these are the only data layers that are currently supported. The data layer cubes incorporate all functionality that is specific to the data layer itself such as generating all the necessary data array composites.
- The bottom of the scheme is the descendant of the data layer cubes referred to as the `BmD_cube`. This class is the one that the end user will interface with as it will contain all functionality related to the generation of the final data product. The functionality that this class will focus on is aimed at the construction of the DataTree and exporting the data

The proposed hierarchy emphasises the modular approach which we consider to be integral towards the development of the final cubing engine. The end product eventually should be able to provide a framework that is flexible in terms of which data cubes can be generated to satisfy a wide range of possible applications.

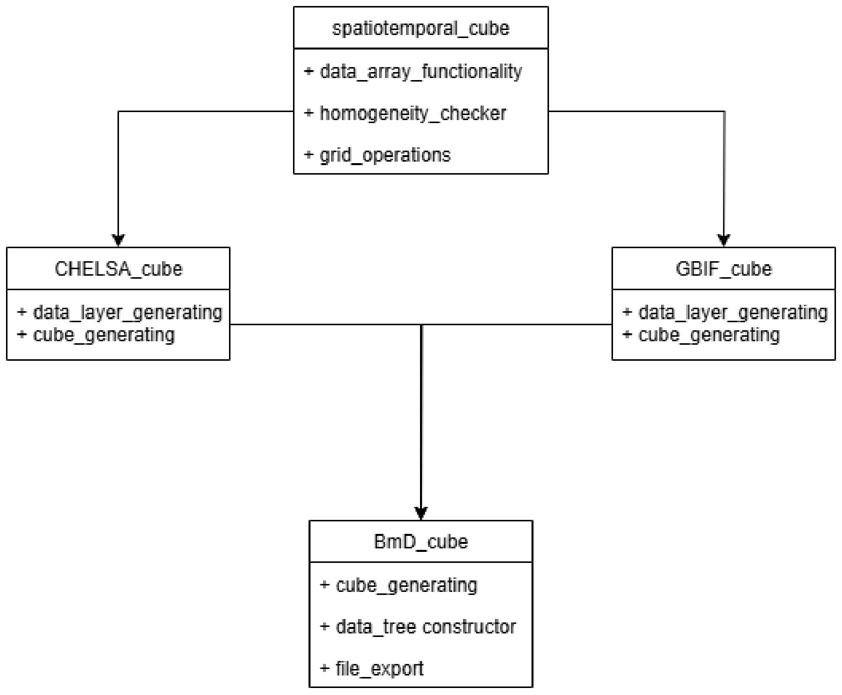


Figure 11: Class hierarchy of the Cubing engine

3.4.2. Spatiotemporal Cube

Base class that is shared across all the data layer cubes and the end product. Groups shared functionality that is universally employed

- Homogenization functions
- Consistency checker functions
- Concatenation functions

The base class `spatiotemporal_cube` serves as the foundation for the other classes that are built on top of it. As such the class contains fundamental functionalities that are rudimentary in the cubing and harmonization process.

- `Da_layer_constructor`, a generalized constructor functions that retrieve slices based on a set of parameters from a data layer using one of the functions defined within the layer submodules in the datasource module. The function returns a composite of all these slices as a list
- `Da_concat`, a concatenation function that takes the result from the constructor function and forms a new data array that combines the list of data arrays. The result is a data array that shares the dimensions of the original data arrays but with a new dimension introduced.
- `Regrid_spatial_coordinates`, a regridding function that currently only performs upscaling of a lower resolution raster to a higher resolution raster. The regridding is performed based on a linear interpolation to fit a raster defined by a target longitude and latitude array. Linear interpolation results in padding being introduced at the borders of the rasters containing NaN values. The standard settings of this function fill these NaN values with the nearest value.

3.4.3. Data Layer Cubes

1.1.1.1. CHELSA Cube

The `chelsa_cube` builds further on the base class by specifically defining layer functions for the different layers present in CHELSA. The general flow of these functions is

1) Extract the requested parameters from the parameter file which is a mandatory function argument
2) Retrieve all the slices from the layer and combine the individual data arrays using the built in `da_layer_constructor` function

For certain slices within layers regridding is required and is currently hard coded in the layer functions themselves. For both the monthly time series and the reference monthly climatological layers the `clt` variable is stored at a lower resolution (roughly 3 fold) than the other variables. In both these layers we perform a regridding with the built in functions.

In future implementations we will aim at automatically identifying resolution inconsistencies and automatic regridding to either the lowest or highest resolution. This needs further discussion with partners from WP5.

1.1.1.1. GBIF Cube

Currently the GBIF cube contains limited functionality. This is mainly due to the cubing being performed by the GBIF API itself. In addition to this the specific data requirements are still being discussed internally between the different work packages of the BMD project.

The main function this class contains is the `construct_gbif_layer` function. This function ingests the parameter file and path and constructs a dataframe from it. Subsequently we index the dataframe according to the relevant dimensions and convert it to a sparse data array. This can be done efficiently due to xarray being built on top of pandas.

As was explained in the GBIF section we perform this transformation due to the large number of dimensions, cardinality and sparseness of the data. This returns a memory efficient representation of the biodiversity data from GBIF

3.4.4. BMD Cube

The BMD cube class is the child of the different data layers combined together (see Fig.12). As such all functionality that is present within these data layer cubes is inherited into this class. On top of this functionality, the cube is equipped with methods that focus on the construction and export of the data tree that combines the different layers together in an overarching data structure.

The root of the data tree is the BMD cube node that branches off into 2 temporal branches. We have chosen to incorporate a `static` branch and a `dynamic` branch. The `static` branch captures all the data layers that describe information that span long periods of time, such as the climatologies, or that are not varying in the time dimension. The `dynamic` branch contains those data layers in which time is a dimension that provides a time series which can be sliced, such as monthly CHELSA data or the occurrence data. Fig.11 shows a visual representation of the current data end product and which dimensions each of these layers possess.

The following methods are included within the `bmd_cube` class

- `generate_bmd_data`, reads the parameter file and automatically parses which data layers should be included into the end product. For each of the enabled layers the appropriate methods that are associated with the data layer cube are called and the data is written to a dictionary that associates it to either the `static` or `dynamic` data group.
- `construct_datatree`, fetches the data that was generated with the `generate_bmd_data` method and constructs a datatree object that is based on the dictionary structure to which the data was written.
- The following two methods were developed to circumvent the limitations of the netcdf file format and its incompatibility with sparse data representation. Netcdf stores data in a dense representation and thus can not store sparse.COO data objects. To circumvent this we implemented methods that convert the sparse data into ‘pseudodense’ representation which allows us to store it in dense format without actually conversion to a dense array.

    - `sparse_to_pseudodense`, takes a dataset and a data variable name and decomposes the sparse.COO object into its components, i.e. the values-indices, coordinates, shape of the array and the dimension names. In addition to this a meta data attribute is created containing the original data variable name. We avoid full dense conversion by storing each structural component of the data array as a variable in a new Dataset object.
    - `load_sparse`, takes a Dataset object that was generated with the previous function and reconstructs the original sparse DataArray. Because we saved not only the values but also the structure schema we can perform this conversion easily.
- `export_tree`, takes the tree that was constructed and iterates over all the branches and leaves of the data to check if any of them are present in sparse format. Any encountered sparse dataset will be converted into a pseudodense representation and subsequently this disc ready data tree is exported to the user's path of choice as a NetCDF file
- `import_tree`, takes the exported datatree and converts it back to a user-ready data format where the pseudodense representation is converted back to a sparse representation.

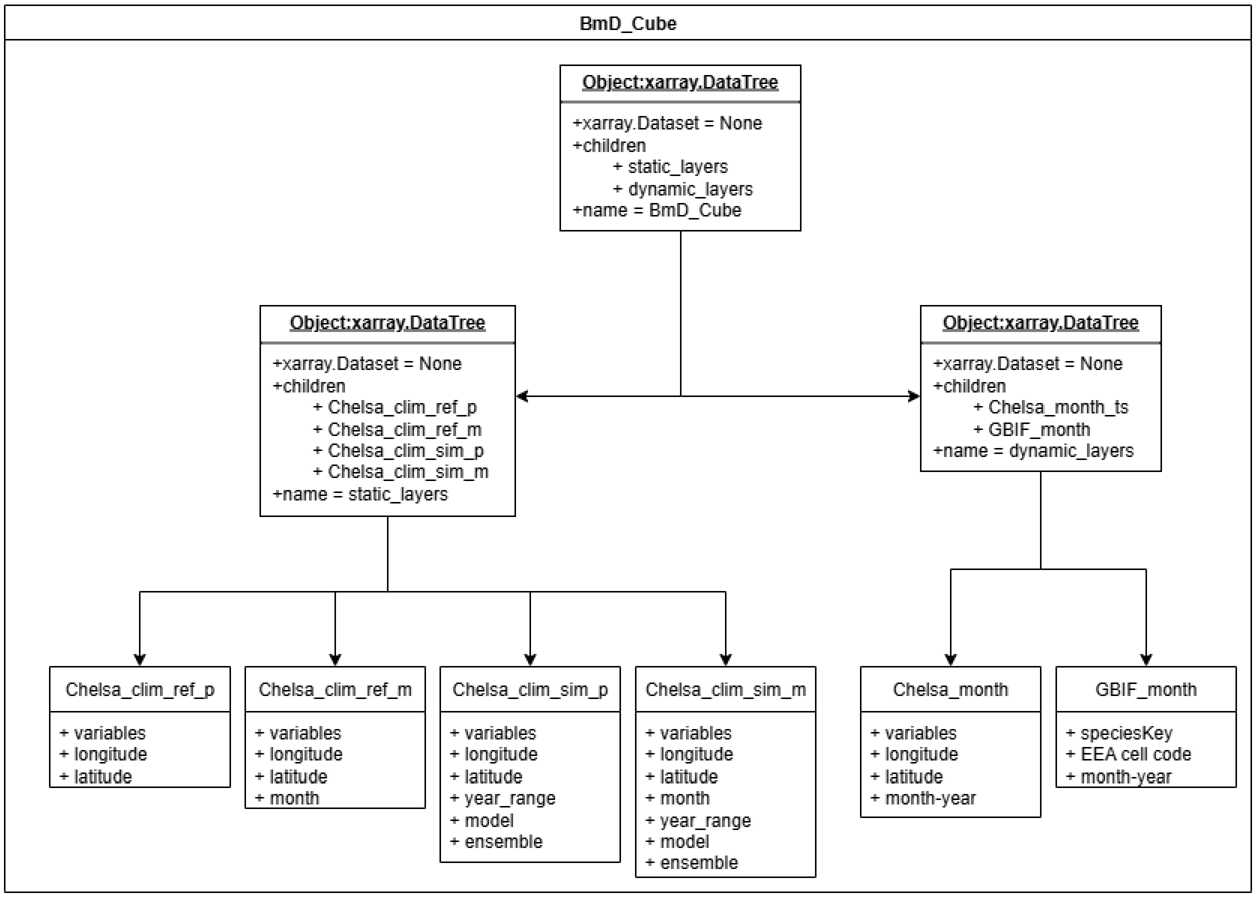


Figure 12: Datatree representation of the BmD cube

3.4.5. Code Usage and input methods

In order to generate a data cube, the library requires a way to efficiently and clearly define the desired parameters for the data that should be requested. As a way to interface with the library we have included a template yaml configuration which can be customized to the data needs of the user. Using a yaml file easily allows us to read in the different parameters in a structured way due to the ease at which it can be deserialized as a python dictionary. Additionally, if the user interface is developed in such a way that the users checks different options through a dashboard for example, it is possible to serialize the selected options to a yaml file that follows the predefined structure.

In terms of the structure of the yaml file itself we make a distinction between the spatial component and the data component. Below we provide both these components and will explain them more in detail.

```yaml
spatial:
  # select the method which should be employed
  method: "bbox"
  bbox:
    long_min: 0
    long_max: 0
    lat_min: 0
    lat_max: 0
  polygon:
    shapefile_path: "/shapefile/filename.shp"
```

The spatial component relates to all the options that determine how and which area we are interested in sampling. We provide 2 ways of sampling data, namely through a bounding box or a polygonal shape. The user should define their desired way of sampling by filling in the method with the appropriate key, i.e. `bbox` or `polygon`.

If one chooses to utilize the `bbox` method it is necessary to provide the minimal coordinates that define the bottom left and the upper right corner of the window. Alternatively if the user chooses to sample through a polygon they need to provide the path that points towards the shapefile where the polygon is stored. Currently polygon sampling has been implemented but has not been used in combination with the construction of the data arrays yet.

```yaml
Layers:
  chelsa_clim_ref_period:
    enabled: true
    time:
      year_range: "1981-2010"
    variables:
      include_all: true
      included: []
      excluded: []
    source:
      base_url:
"https://os.zhdk.cloud.switch.ch/chelsav2/GLOBAL/climatologies/1981-2010"
      version: "V.2.1"
```

In the layers component we specify which data layers are requested and, if they are requested, which parameters of the data layer in question needs to be sampled. All data layers follow a similar scheme in how they are structured and above we give the example of the monthly CHELSA climate reference over a period layer due to the fact that it is the smallest and easiest to understand.

All layers adhere to the following blueprint

- The `enabled` keyword is a boolean that toggles the inclusion of the layer in the end data cube. Possible values for this keyword are `true` or `false`
- Subsequent keywords deal with dimensions that characterize the data layer. Current dimension key words are `time`, `variables`, `model` and `ensemble`. Generally all these dimensions keywords follow a set structure, though some may vary due to the nature of how the data layer function is defined.
    - `include_all` keyword toggles the inclusion of all possible values for the given dimension and is characterized by a boolean. E.g. if we toggle this to `true` for the `variables` keyword all variables within this data layer will be included in the final data cube
    - `included` keyword is only utilized when `include_all` is set to false. This keyword is a list in which all options of the dimension that should be included while the rest will be disregarded. E.g. if we have a data layer that spans the years [2020, 2021, 2022, 2023] and the `included` keyword contains [2020, 2022] then the other years will be disregarded and only these 2 will be included in the end product.
    - `excluded` keyword similarly only gets utilized when `include_all` is set to `false`. The behavior of this keyword is opposite to the `included` one and will take all possible options of the dimensions barring the ones mentioned in this list. E.g. using the example from the `included` keyword if we specify the `excluded` list as [2023] than the cube will generated using [2020, 2021, 2022]
- Optional keywords already have values associated with them and should normally not be modified by the end user in some cases. The source keyword groups together all the options that are related to the address of the S3 bucket and the version of CHELSA that is currently being used.

4. End product

The code that was written can be found in the BmC Github repository. The main branch contains all the necessary code to execute cube generation itself. In addition to this we have also provided a jupyter notebook called `data_generation` in the `scripts/prototype` directory that will guide the user through using the cubing engine prototype.

An example of the cubing engine output can be found in the output directory under prototype_cube. This is a very limited version of a data cube that was utilized during development of the cubing engine and can be included in the repository due to its small size.

In order to actually generate cubes that are more realistic we will provide some example parameter yaml files within the config so that they can be generated locally.

5. Future considerations and issues

The current prototype provides a basis on which we can build further. We have provided functionality that fetches data from biotic and abiotic data sources and have devised a schema for the structuring of the different layer components to the end cube product.

While exploring this we have stumbled on some key take-aways and features that we will need to consider when developing the end product.

- Further functionality in transforming and processing the data will be required. The current sources do not require extensive transformation but we anticipate that in order to make other data sources compatible we will need to implement functionality that is aimed specifically at homogenization of the data. Functions that bring everything to a shared CRS comes to mind but also regridding functions
- The data needs with regards to biodiversity data have not been cleared up completely. In this report we have highlighted the difference between vectorized and rasterized data. Questions in regard to how we best represent this data to be used by the VREs remain and will require input from stakeholders and partners to clearly shape what the end product should look like.
- Additional functionality to up- and down-sample ingested data needs to be provided as well, such as moving window computations. In addition to this it will also be necessary to further expand upon the structure of the datatree and the parameter file to accommodate these features.
- The current prototype has not yet implemented meta data into the cube and this must definitely be taken into account to be compliant with the FAIR principles. This includes metadata relating to the sampling protocol used to acquire the occurrence data.

In addition to these future considerations we will start to focus on the integration of additional abiotic data sources within the cubing engine. This will happen in cooperation with WP2 who are involved in the development of the data catalogue. We expect that the next source of data will be the WeKEO service that will provide us access to important data layers such as land usage, elevation models, etc.

6. References

Hoyer, S. and Hamman, J. (2017) 'xarray: N-D labeled Arrays and Datasets in Python', *Journal of Open Research Software*, 5(1), p. 10. Available at: https://doi.org/10.5334/jors.148.

https://www.geeksforgeeks.org/java/multidimensional-arrays-in-java/
https://r-spatial.org/book/06-Cubes_files/figure-html/fig-cube-1.png
https://r-tmap.github.io/tmap-book/geodata.html
https://cdn.infodiagram.com/c/4761d3/operations-olap-cube.png
https://tutorial.xarray.dev/fundamentals/01_data_structures.html
https://media.geeksforgeeks.org/wp-content/uploads/Sparse-Matrix-Array-Representation1.png