Package 'arealDB'

January 20, 2025

Title Harmonise and Integrate Heterogeneous Areal Data

Description Many relevant applications in the environmental and socioeconomic sciences use areal data, such as biodiversity checklists, agricultural statistics, or socioeconomic surveys. For applications that surpass the spatial, temporal or thematic scope of any single data source, data must be integrated from several heterogeneous sources. Inconsistent concepts, definitions, or messy data tables make this a tedious and error-prone process. 'arealDB' tackles those problems and helps the user to integrate a harmonised databases of areal data. Read the paper at Ehrmann, Seppelt & Meyer (2020) <doi:10.1016/j.envsoft.2020.104799>.

Version 0.9.4

URL https://github.com/luckinet/arealDB

BugReports https://github.com/luckinet/arealDB/issues

Depends R (>= 3.5.0)

Imports archive, beepr, checkmate, dplyr, fuzzyjoin, magrittr, ontologics, progress, purrr, readr, rlang, rmapshaper, stringr, sf, tabshiftr, tibble, tidyr, tidyselect,

Suggests testthat, knitr, rmarkdown, bookdown, covr

Language en-gb

License GPL-3

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

VignetteBuilder knitr

NeedsCompilation no

Author Steffen Ehrmann [aut, cre] (<https://orcid.org/0000-0002-2958-0796>),
Arne Rümmler [aut, ctb] (<https://orcid.org/0000-0001-8637-9071>),
Felipe Melges [ctb] (<https://orcid.org/0000-0003-0833-8973>),
Carsten Meyer [aut] (<https://orcid.org/0000-0003-3927-5856>)

Maintainer Steffen Ehrmann <steffen.ehrmann@posteo.de>

Repository CRAN

Date/Publication 2025-01-20 13:40:05 UTC

Contents

`.editMatches` *Edit matches manually in a csv-table*

Description

Allows the user to match concepts with an already existing ontology, without actually writing into the ontology, but instead storing the resulting matching table as csv.

Usage

```
.editMatches(
  new,
  topLevel,
  source = NULL,
  ontology = NULL,
  matchDir = NULL,
  stringdist = TRUE,
  verbose = TRUE,
  beep = NULL
)
```

Arguments

`new`	`data.frame(.)` the new concepts that shall be manually matched, includes "label", "class" and "has_broader" columns.
`topLevel`	`logical(1)` whether or not the new concepts are at the highest level only, i.e., have to be matched without context, or whether they are contain columns that must be matched within parent columns.
`source`	`character(1)` any character uniquely identifying the source dataset of the new concepts.
`ontology`	`ontology(1)` either a path where the ontology is stored, or an already loaded ontology.
`matchDir`	`character(1)` the directory where to store source-specific matching tables.
`stringdist`	`logical(1)` whether or not to use string distance to find matches (should not be used for large datasets/when a memory error is shown).
`verbose`	`logical(1)` whether or not to give detailed information on the process of this function.
`beep`	`integerish(1)` Number specifying what sound to be played to signal the user that a point of interaction is reached by the program, see `beep`.

Details

In order to match new concepts into an already existing ontology, it may become necessary to carry out manual matches of the new concepts with already harmonised concepts, for example, when the new concepts are described with terms that are not yet in the ontology. This function puts together a table, in which the user would edit matches by hand. Whith the argument `verbose = TRUE`, detailed information about the edit process are shown to the user. After defining matches, and even if not all necessary matches are finished, the function stores a specific "matching table" with the name *match_SOURCE.csv* in the respective directory (`matchDir`), from where work can be picked up and continued at another time.

Fuzzy matching is carried out and matches with 0, 1 or 2 differing charcters are presented in a respective column.

Value

A table that contains all new matches, or if none of the new concepts weren't already in the ontology, a table of the already sucessful matches.

`.getColTypes` *Get the column types of a tibble*

Description

(internal function not for user interaction)

Usage

```
.getColTypes(input = NULL)
```

Arguments

`input` data.frame
table from which to get column types.

`.matchOntology` *Match target terms with an ontology*

Description

This function takes a table to replace the values of various columns with harmonised values listed in the project specific gazetteer.

Usage

```
.matchOntology(
  table = NULL,
  columns = NULL,
  dataseries = NULL,
  ontology = NULL,
  beep = NULL,
  colsAsClass = TRUE,
  groupMatches = FALSE,
  stringdist = TRUE,
  strictMatch = FALSE,
  verbose = FALSE
)
```

Arguments

`table` `data.frame(1)`
a table that contains columns that should be harmonised by matching with the gazetteer.

`columns` `character(1)`
the columns containing the concepts

`dataseries` `character(1)`
the source dataseries from which territories are sourced.

`ontology` `onto`
path where the ontology/gazetteer is stored.

`beep` `integerish(1)`
Number specifying what sound to be played to signal the user that a point of interaction is reached by the program, see `beep`.

`colsAsClass` `logical(1)`
whether to match `columns` by their name with the respective classes, or with concepts of all classes.

`groupMatches` `logical(1)`
whether or not to group harmonized concepts when there are more than one match (for example for broader or narrower matches).

`stringdist` `logical(1)`
whether or not to use string distance to find matches (should not be used for large datasets/when a memory error is shown).

`strictMatch` `logical(1)`
whether or not matches are strict, i.e., there should be clear one-to-one relationships and no changes in broader concepts.

`verbose` `logical(1)`
whether or not to give detailed information on the process of this function.

Value

Returns a table that resembles the input table where the target columns were translated according to the provided ontology.

`.updateOntology` *Update an ontology*

Description

This function takes a table (spatial) and updates all territorial concepts in the provided gazetteer.

Usage

```
.updateOntology(
  table = NULL,
  threshold = NULL,
  dataseries = NULL,
  ontology = NULL
)
```

Arguments

table
: character(1)
: a table that contains a match column as the basis to update the gazetteer.

threshold
: numeric(1)
: a threshold value above which matches are updated in the gazetteer.

dataseries
: character(1)
: the source dataseries of the external concepts for which the gazetteer shall be updated.

ontology
: onto
: path where the ontology/gazetteer is stored.

Value

called for its side-effect of updating a gazetteer

---

adb_archive *Archive the data from an areal database*

Description

Archive the data from an areal database

Usage

```
adb_archive(pattern = NULL, variables = NULL, compress = FALSE, outPath = NULL)
```

Arguments

pattern
: character(1)
: a regular expression used to filter files to load.

variables
: character(.)
: columns, typically observed variables, to select.

compress
: logical(1)
: whether or not the database should be compressed into a *tar.gz* archive. Will delete the database folder in `outPath`.

outPath
: character(1)
: directory, where the archive should be stored.

Details

This function prepares and packages the data into an archiveable form. This contains geopacakge files for geometries and csv files for all tables, such as inventory, matching and thematic data tables.

Value

no return value, called for the side-effect of creating a database archive.

`adb_backup` *Backup the current state of an areal database*

Description

Backup the current state of an areal database

Usage

```
adb_backup()
```

Details

This function creates a tag that is composed of the version and the date, appends it to all stage3 files (tables and geometries), the inventory and the ontology/gazetteer files and stores them in the backup folder of the current areal database.

Value

No return value, called for the side effect of saving the inventory, the stage3 files and modified ontology/gazetteer into the backup directory.

`adb_diagnose` *Diagnose databse contents*

Description

work in progress, not yet useable

Usage

```
adb_diagnose(
  territory = NULL,
  concept = NULL,
  variable = NULL,
  level = NULL,
  year = NULL
)
```

Arguments

`territory`	description
`concept`	description
`variable`	description
`level`	description
`year`	description

adb_example *Build an example areal database*

Description

This function helps setting up an example database up until a certain step.

Usage

```
adb_example(path = NULL, until = NULL, verbose = FALSE)
```

Arguments

path `character(1)`
The database gets created by default in tempdir(), but if you want it in a particular location, specify that in this argument.

until `character(1)`
The database building step in terms of the function names until which the example database shall be built, one of `"start_arealDB"`, `"regDataseries"`, `"regGeometry"`, `"regTable"`, `"normGeometry"` or `"normTable"`.

verbose `logical(1)`
be verbose about building the example database (default `FALSE`).

Details

Setting up a database with an R-based tool can appear to be cumbersome and too complex and thus intimidating. By creating an example database, this functions allows interested users to learn step by step how to build a database of areal data. Moreover, all functions in this package contain verbose information and ask for information that would be missing or lead to an inconsistent database, before a failure renders hours of work useless.

Value

No return value, called for the side effect of creating an example database at the specified `path`.

Examples

```
if(dev.interactive()){
# to build the full example database
adb_example(path = paste0(tempdir(), "/newDB"))

# to make the example database until a certain step
adb_example(path = paste0(tempdir(), "/newDB"), until = "regDataseries")

}
```

adb_init *Initiate an areal database*

Description

Initiate a geospatial database or register a database that exists at the root path.

Usage

```
adb_init(
  root,
  version,
  author,
  licence,
  ontology,
  gazetteer = NULL,
  top = NULL,
  staged = TRUE
)
```

Arguments

root
character(1)
path to the root directory that contains or shall contain an areal database.

version
character(1)
version identifier for this areal database.

author
character(1)
authors that contributed to building this areal database. Should be a list with items "cre" (creator), "aut" (authors) and "ctb" (contributors).

licence
character(1)
licence (link) for this areal database.

ontology
list(.)
named list with the path(s) of ontologies, where the list name identifies the variable that shall be matched with the ontology at the path.

gazetteer
character(1)
path to the gazetteer that holds the (hierarchical) information of territorial units used in this database.

top
character(1)
the label of the class in the gazetteer that represents the top-most unit (e.g. country) of the areal database that shall be started.

staged
logical(1)
whether or not the file structure is arranged according to stages (with geometries and tables separated), or merely as input/output (of all types).

Details

This is the first function that is run in a project, as it initiates the areal database by creating the default sub-directories and initial inventory tables. When a database has already been set up, this function is used to register that path in the options of the current R session.

Value

No return value, called for the side effect of creating the directory structure of the new areal database and tables that contain the database metadata.

Examples

```
adb_init(root = paste0(tempdir(), "/newDB"),
         version = "1.0.0", licence = "CC-BY-0.4",
         author = list(cre = "Gordon Freeman", aut = "Alyx Vance", ctb = "The G-Man"),
         gazetteer = paste0(tempdir(), "/newDB/territories.rds"),
         top = "al1",
         ontology = list(var = paste0(tempdir(), "/newDB/ontology.rds")))

getOption("adb_path"); getOption("gazetteer_path")
```

`adb_inventory` *Load the inventory of the currently active areal database*

Description

Load the inventory of the currently active areal database

Usage

```
adb_inventory(type = NULL)
```

Arguments

type	`character(1)` the inventory sub-table to load, either `"dataseries"`, `"tables"`, `"geometries"` or `"references"`.

Value

returns the table selected in `type`

`adb_metadata` *Load the metadata from an areal database*

Description

Load the metadata from an areal database

Usage

```
adb_metadata()
```

`adb_ontology` *Load the currently active ontology*

Description

Load the currently active ontology

Usage

```
adb_ontology(..., type = "ontology")
```

Arguments

`...`	combination of column name in the ontology and value to filter that column by to build a tree of the concepts nested into it; see `make_tree`.
`type`	`character(1)` the type of ontology to load, either `"ontology"` to get the thematic concepts, or `"gazetteer"` to get the territories.

Value

returns a tidy table of an ontology or gazetteer that is used in an areal database.

adb_querry *Extract database contents*

Description

Extract database contents

Usage

```
adb_querry(
  territory = NULL,
  concept = NULL,
  variable = NULL,
  level = NULL,
  year = NULL
)
```

Arguments

territory	‘character(.) combination of column name in the ontology and value to filter that column by to build a tree of the territories nested into it.
concept	description
variable	description
level	description
year	description

Value

returns ...

Examples

```
if(dev.interactive()){
adb_example(path = paste0(tempdir(), "/newDB"))

adb_querry(territory = list(al1 = "a_nation"),
           concept = list(commodity = "barley"),
           variable = "harvested")
}
```

adb_reset *Reset an areal database to its unfilled state*

Description

Reset an areal database to its unfilled state

Usage

```
adb_reset(what = "all")
```

Arguments

what	logical(1) what to reset, either "onto", "gaz", "schemas", "tables", "geometries" or "all", the default.

Value

no return value, called for its side effect of reorganising an areal database into a state where no reg* or norm* functions have been run

adb_restore *Restore the database from a backup*

Description

Restore the database from a backup

Usage

```
adb_restore(version = NULL, date = NULL)
```

Arguments

version	‘character(1) a version tag for which to restore files.
date	character(1) a date for which to restore files.

Details

This function searches for files that have the version and date tag, as it was defined in a previous run of adb_backup, to restore them to their original folders. This function overwrites by default, so use with care.

Value

No return value, called for the side effect of restoring files that were previously stored in a backup.

`adb_schemas` *Load the schemas of the currently active areal database*

Description

Load the schemas of the currently active areal database

Usage

```
adb_schemas(pattern = NULL)
```

Arguments

`pattern` `character(1)`
an optional regular expression. Only schema names which match the regular expression will be processed.

Value

returns a list of schema descriptions

`adb_translations` *Load the translation tables of the currently active areal database*

Description

Load the translation tables of the currently active areal database

Usage

```
adb_translations(type = NULL, dataseries = NULL)
```

Arguments

`type` `character(1)`
the type of ontology for which to load translation tables, either `"ontology"` to get the thematic concepts, or `"gazetteer"` to get the territories.

`dataseries` `character(1)`
the name of a dataseries as registered in `regDataseries`.

Value

returns the selected translation table

`normGeometry` *Normalise geometries*

Description

Harmonise and integrate geometries into a standardised format

Usage

```
normGeometry(
  input = NULL,
  pattern = NULL,
  query = NULL,
  thresh = 10,
  beep = NULL,
  simplify = FALSE,
  stringdist = TRUE,
  strictMatch = FALSE,
  verbose = FALSE
)
```

Arguments

`input` `character(1)`
path of the file to normalise. If this is left empty, all files at stage two as subset by `pattern` are chosen.

`pattern` `character(1)`
an optional regular expression. Only dataset names which match the regular expression will be processed.

`query` `character(1)`
part of the SQL query (starting from WHERE) used to subset the input geometries, for example `"where NAME_0 = 'Estonia'"`. The first part of the query (where the layer is defined) is derived from the meta-data of the currently handled geometry.

`thresh` `integerish(1)`
percent value of overlap below which two geometries (the input and the base) are considered to be the same. This is required, because often the polygons from different sources, albeit describing the same territorial unit, aren't completely the same.

`beep` `integerish(1)`
Number specifying what sound to be played to signal the user that a point of interaction is reached by the program, see `beep`.

`simplify` `logical(1)`
whether or not to simplify geometries.

`stringdist` `logical(1)`
whether or not to use string distance to find matches (should not be used for large datasets/when a memory error is shown).

`strictMatch` `logical(1)`
whether or not matches are strict, i.e., there should be clear one-to-one relationships and no changes in broader concepts.

`verbose` `logical(1)`
be verbose about what is happening (default `FALSE`). Furthermore, you can use `suppressMessages` to make this function completely silent.

Details

To normalise geometries, this function proceeds as follows:

1. Read in `input` and extract initial metadata from the file name.
2. In case filters are set, the new geometry is filtered by those.
3. The territorial names are matched with the gazetteer to harmonise new territorial names (at this step, the function might ask the user to edit the file 'matching.csv' to align new names with already harmonised names).
4. Loop through every nation potentially included in the file that shall be processed and carry out the following steps:
    - In case the geometries are provided as a list of simple feature POLYGONS, they are dissolved into a single MULTIPOLYGON per main polygon.
    - In case the nation to which a geometry belongs has not yet been created at stage three, the following steps are carried out:
        (a) Store the current geometry as basis of the respective level (the user needs to make sure that all following levels of the same dataseries are perfectly nested into those parent territories, for example by using the GADM dataset)
    - In case the nation to which the geometry belongs has already been created, the following steps are carried out:
        (a) Check whether the new geometries have the same coordinate reference system as the already existing database and re-project the new geometries if this is not the case.
        (b) Check whether all new geometries are already exactly matched spatially and stop if that is the case.
        (c) Check whether the new geometries are all within the already defined parents, and save those that are not as a new geometry.
        (d) Calculate spatial overlap and distinguish the geometries into those that overlap with more and those with less than `thresh`.
        (e) For all units that dName match, copy gazID from the geometries they overlap.
        (f) For all units that dName not match, rebuild metadata and a new gazID.
    - store the processed geometry at stage three.
5. Move the geometry to the folder '/processed', if it is fully processed.

Value

This function harmonises and integrates so far unprocessed geometries at stage two into stage three of the geospatial database. It produces for each main polygon (e.g. nation) in the registered geometries a spatial file of the specified file-type.

See Also

Other normalise functions: normTable()

Examples

```
if(dev.interactive()){
  library(sf)

  # build the example database
  adb_example(until = "regGeometry", path = tempdir())

  # normalise all geometries ...
  normGeometry(pattern = "estonia")

  # ... and check the result
  st_layers(paste0(tempdir(), "/geometries/stage3/Estonia.gpkg"))
  output <- st_read(paste0(tempdir(), "/geometries/stage3/Estonia.gpkg"))
}
```

normTable *Normalise data tables*

Description

Harmonise and integrate data tables into standardised format

Usage

```
normTable(
  input = NULL,
  pattern = NULL,
  query = NULL,
  ontoMatch = NULL,
  beep = NULL,
  verbose = FALSE
)
```

Arguments

input character(1)
path of the file to normalise. If this is left empty, all files at stage two as subset by pattern are chosen.

pattern character(1)
an optional regular expression. Only dataset names which match the regular expression will be processed.

query character(1)
the expression that would be used in filter to subset a tibble in terms of the columns defined via the schema and given as a single character string, such as "al1 == 'Estonia'".

ontoMatch character(.)
name of the column(s) that shall be matched with an ontology (defined in adb_init).

beep integerish(1)
Number specifying what sound to be played to signal the user that a point of interaction is reached by the program, see beep.

verbose logical(1)
be verbose about translating terms (default FALSE). Furthermore, you can use suppressMessages to make this function completely silent.

Details

To normalise data tables, this function proceeds as follows:

1. Read in input and extract initial metadata from the file name.
2. Employ the function tabshiftr::reorganise() to reshape input according to the respective schema description.
3. The territorial names are matched with the gazetteer to harmonise new territorial names (at this step, the function might ask the user to edit the file 'matching.csv' to align new names with already harmonised names).
4. Harmonise territorial unit names.
5. store the processed data table at stage three.

Value

This function harmonises and integrates so far unprocessed data tables at stage two into stage three of the areal database. It produces for each main polygon (e.g. nation) in the registered data tables a file that includes all thematic areal data.

See Also

Other normalise functions: normGeometry()

Examples

```
if(dev.interactive()){
  # build the example database
  adb_example(until = "normGeometry", path = tempdir())

  # normalise all available data tables ...
  normTable()
  # ... and check the result
  output <- readRDS(paste0(tempdir(), "/tables/stage3/Estonia.rds"))
}
```

`regDataseries` *Register a new dataseries*

Description

This function registers a new dataseries of both, geometries or areal data into the geospatial database. This contains the name and relevant meta-data of a dataseries to enable provenance tracking and reproducability.

Usage

```
regDataseries(
  name = NULL,
  description = NULL,
  homepage = NULL,
  version = NULL,
  licence_link = NULL,
  reference = NULL,
  notes = NULL,
  overwrite = FALSE
)
```

Arguments

`name` `character(1)`
the dataseries abbreviation or name.

`description` `character()`
the "long name" or "brief description" of the dataseries.

`homepage` `character(1)`
the homepage of the data provider where the dataseries or additional information can be found.

`version` `character(1)`
the version number or date when meta data of the dataseries were recorded.

`licence_link` `character(1)`
link to the licence or the webpage from which the licence was copied.

`reference` `bibentry(1)`
in case the dataseries comes with a reference, provide this here as bibentry object.

`notes` `character(1)`
optional notes.

`overwrite` `logical(1)`
whether or not the dataseries to register shall overwrite a potentially already existing older version.

Value

Returns a tibble of the new entry that is appended to 'inv_dataseries.csv'.

See Also

Other register functions: `regGeometry()`, `regTable()`

Examples

```
if(dev.interactive()){
  # start the example database
  adb_exampleDB(until = "match_gazetteer", path = tempdir())

  regDataseries(name = "gadm",
                description = "Database of Global Administrative Areas",
                version = "3.6",
                homepage = "https://gadm.org/index.html",
                licence_link = "https://gadm.org/license.html")
}
```

regGeometry *Register a new geometry entry*

Description

This function registers a new geometry of territorial units into the geospatial database.

Usage

```
regGeometry(
  ...,
  subset = NULL,
  gSeries = NULL,
  label = NULL,
  ancillary = NULL,
  layer = NULL,
  archive = NULL,
  archiveLink = NULL,
  downloadDate = NULL,
  updateFrequency = NULL,
  notes = NULL,
  overwrite = FALSE
)
```

Arguments

`...`
: `character(1)`
optional named argument selecting the main territory into which this geometry is nested. The name of this must be a class of the gazetteer and the value must be one of the territory names of that class, e.g. *nation = "Estonia"*.

`subset`
: `character(1)`
optional argument to specify which subset the file contains. This could be a subset of territorial units (e.g. only one municipality) or of a target variable.

`gSeries`
: `character(1)`
the name of the geometry dataseries (see `regDataseries`).

`label`
: `list(.)`
list of as many columns as there are in common in the ontology and this geometry. Must be of the form `list(class = columnName)`, with 'class' as the class of the ontology corresponding to the respective column name in the geometry.

`ancillary`
: `list(.)`
optinal list of columns containing ancillary information. Must be of the form `list(attribute = columnName)`, where `attribute` can be one or several of

- `"name_ltn"` (the english name in latin letters)
- `"name_lcl"` (the name in local language and letters)
- `"code"` (any code describing the unit)
- `"type"` (the type of territorial unit)
- `"uri"` (the semantic web URI) or
- `"flag"` (any flag attributed to the unit).

`layer`
: `character(1)`
the name of the file's layer from which the geometry should be created (if applicable).

`archive`
: `character(1)`
the original file (perhaps a *.zip) from which the geometry emerges.

`archiveLink`
: `character(1)`
download-link of the archive.

`downloadDate`
: `character(1)`
value describing the download date of this dataset (in YYYY-MM-DD format).

`updateFrequency`
: `character(1)`
value describing the frequency with which the dataset is updated, according to the ISO 19115 Codelist, MD_MaintenanceFrequencyCode. Possible values are: 'continual', 'daily', 'weekly', 'fortnightly', 'quarterly', 'biannually', 'annually', 'asNeeded', 'irregular', 'notPlanned', 'unknown', 'periodic', 'semimonthly', 'biennially'.

`notes`
: `character(1)`
optional notes that are assigned to all features of this geometry.

`overwrite`
: `logical(1)`
whether or not the geometry to register shall overwrite a potentially already existing older version.

Details

When processing geometries to which areal data shall be linked, carry out the following steps:

1. Determine the main territory (such as a nation, or any other polygon), a `subset` (if applicable), the dataseries of the geometry and the ontology `label`, and provide them as arguments to this function.
2. Run the function.
3. Export the shapefile with the following properties:
   - Format: GeoPackage
   - File name: What is provided as message by this function
   - CRS: EPSG:4326 - WGS 84
   - make sure that 'all fields are exported'
4. Confirm that you have saved the file.

Value

Returns a tibble of the entry that is appended to 'inv_geometries.csv'.

See Also

Other register functions: `regDataseries()`, `regTable()`

Examples

```
if(dev.interactive()){
  # build the example database
  adb_exampleDB(until = "regDataseries", path = tempdir())

  # The GADM dataset comes as *.7z archive
  regGeometry(gSeries = "gadm",
              label = list(al1 = "NAME_0"),
              layer = "example_geom1",
              archive = "example_geom.7z|example_geom1.gpkg",
              archiveLink = "https://gadm.org/",
              nextUpdate = "2019-10-01",
              updateFrequency = "quarterly")

  # The second administrative level in GADM contains names in the columns
  # NAME_0 and NAME_1
  regGeometry(gSeries = "gadm",
              label = list(al1 = "NAME_0", al2 = "NAME_1"),
              ancillary = list(name_lcl = "VARNAME_1", code = "GID_1", type = "TYPE_1"),
              layer = "example_geom2",
              archive = "example_geom.7z|example_geom2.gpkg",
              archiveLink = "https://gadm.org/",
              nextUpdate = "2019-10-01",
              updateFrequency = "quarterly")
}
```

regTable *Register a new areal data table*

Description

This function registers a new areal data table into the geospatial database.

Usage

```
regTable(
  ...,
  subset = NULL,
  dSeries = NULL,
  gSeries = NULL,
  label = NULL,
  begin = NULL,
  end = NULL,
  schema = NULL,
  archive = NULL,
  archiveLink = NULL,
  downloadDate = NULL,
  updateFrequency = NULL,
  metadataLink = NULL,
  metadataPath = NULL,
  notes = NULL,
  diagnose = FALSE,
  overwrite = FALSE
)
```

Arguments

`...` `character(1)`
name and value of the topmost unit under which the table shall be registered. The name of this must be a class of the gazetteer and the value must be one of the territory names of that class, e.g. *nation = "Estonia"*.

`subset` `character(1)`
optional argument to specify which subset the file contains. This could be a subset of territorial units (e.g. only one municipality) or of a target variable.

`dSeries` `character(1)`
the dataseries of the areal data (see `regDataseries`).

`gSeries` `character(1)`
optionally, the dataseries of the geometries, if the geometry dataseries deviates from the dataseries of the areal data (see `regDataseries`).

`label` `integerish(1)`
the label in the onology this geometry should correspond to.

`begin` `integerish(1)`
the date from which on the data are valid.

`end` `integerish(1)`
the date until which the data are valid.

`schema` `schema`
the schema description of the table to read in (must have been placed in the global environment before calling it here).

`archive` `character(1)`
the original file from which the boundaries emerge.

`archiveLink` `character(1)`
download-link of the archive.

`downloadDate` `character(1)`
value describing the download date of this dataset (in YYYY-MM-DD format).

`updateFrequency`
`character(1)`
value describing the frequency with which the dataset is updated, according to the ISO 19115 Codelist, MD_MaintenanceFrequencyCode. Possible values are: ’continual’, ’daily’, ’weekly’, ’fortnightly’, ’quarterly’, ’biannually’, ’annually’, ’asNeeded’, ’irregular’, ’notPlanned’, ’unknown’, ’periodic’, ’semi-monthly’, ’biennially’.

`metadataLink` `character(1)`
if there is already metadata existing: link to the meta dataset.

`metadataPath` `character(1)`
if an existing meta dataset was downloaded along the data: the path where it is stored locally.

`notes` `character(1)`
optional notes.

`diagnose` `logical(1)`
whether or not to try to reorganise the table with the provided schema. note: this does not save the reogranised table into the database yet, further steps of harmonisation are carried out by `normTable` before that.

`overwrite` `logical(1)`
whether or not the geometry to register shall overwrite a potentially already existing older version.

Details

When processing areal data tables, carry out the following steps:

1. Determine the main territory (such as a nation, or any other polygon), a `subset` (if applicable), the ontology `label` and the dataseries of the areal data and of the geometry, and provide them as arguments to this function.
2. Provide a `begin` and `end` date for the areal data.
3. Run the function.
4. (Re)Save the table with the following properties:

    - Format: csv
    - Encoding: UTF-8
    - File name: What is provided as message by this function
    - make sure that the file is not modified or reshaped. This will happen during data normalisation via the schema description, which expects the original table.
5. Confirm that you have saved the file.

Every areal data dataseries (`dSeries`) may come as a slight permutation of a particular table arrangement. The function `normTable` expects internally a schema description (a list that describes the position of the data components) for each data table, which is saved as `paste0("meta_", dSeries, TAB_NUMBER)`. See package `tabshiftr`.

Value

Returns a tibble of the entry that is appended to 'inv_tables.csv' in case `update` = `TRUE`.

See Also

Other register functions: `regDataseries()`, `regGeometry()`

Examples

```
if(dev.interactive()){
  # build the example database
  adb_exampleDB(until = "regGeometry", path = tempdir())

  # the schema description for this table
  library(tabshiftr)

  schema_madeUp <-
    setIDVar(name = "al1", columns = 1) %>%
    setIDVar(name = "year", columns = 2) %>%
    setIDVar(name = "commodities", columns = 3) %>%
    setObsVar(name = "harvested",
              factor = 1, columns = 4) %>%
    setObsVar(name = "production",
              factor = 1, columns = 5)

  regTable(nation = "Estonia",
           subset = "barleyMaize",
           label = "al1",
           dSeries = "madeUp",
           gSeries = "gadm",
           begin = 1990,
           end = 2017,
           schema = schema_madeUp,
           archive = "example_table.7z|example_table1.csv",
           archiveLink = "...",
           nextUpdate = "2024-10-01",
           updateFrequency = "quarterly",
           metadataLink = "...",
            metadataPath = "my/local/path")
}
```

territories *Example* gazetteer

Description

An ontology of territory names (gazetteer)

Usage

```
territories
```

Format

object of class onto for the example territories used in adb_example.

Index