Package 'autodb'

March 19, 2025

Title Automatic Database Normalisation for Data Frames

Version 2.3.1

Description Automatic normalisation of a data frame to third normal form, with the intention of easing the process of data cleaning. (Usage to design your actual database for you is not advised.)
Originally inspired by the 'AutoNormalize' library for 'Python' by 'Alteryx' (<https://github.com/alteryx/autonormalize>), with various changes and improvements. Automatic discovery of functional or approximate dependencies, normalisation based on those, and plotting of the resulting ``database'' via 'Graphviz', with options to exclude some attributes at discovery time, or remove discovered dependencies at normalisation time.

License BSD_3_clause + file LICENSE

Encoding UTF-8

Language en-GB

RoxygenNote 7.3.2

Depends R (>= 4.1.0)

Imports rlang

Suggests spelling, DiagrammeR (>= 1.0.7), testthat (>= 3.1.4), R.utils (>= 2.11.0), hedgehog (>= 0.1), tibble(>= 3.2.1), knitr, rmarkdown

VignetteBuilder knitr

LazyData true

URL https://charnelmouse.github.io/autodb/, https://github.com/CharnelMouse/autodb

BugReports https://github.com/CharnelMouse/autodb/issues

NeedsCompilation no

Author Mark Webster [aut, cre] (<https://orcid.org/0000-0002-3351-0686>)

Maintainer Mark Webster <markwebster204@yahoo.co.uk>

Repository CRAN

Date/Publication 2025-03-19 12:40:02 UTC

Contents

attrs *Relational data attributes*

Description

Generic function, for fetching attribute sets for elements of a relational object.

Usage

```
attrs(x, ...)

attrs(x, ...) <- value
```

Arguments

x	a relational schema object, such as a relation_schema or database_schema object, or a relational data object, such as a relation or database object.
...	further arguments passed on to methods.
value	A character vector of the same length as attrs(x, ...).

Value

A list, containing a character vector for each element of x.

attrs_order *Relational data attribute order*

Description

Generic function, fetching attribute order for relational objects.

Usage

```
attrs_order(x, ...)

attrs_order(x, ...) <- value
```

Arguments

x	an R object, such as a functional_dependency, relation_schema, relation, database_schema, or database object.
...	further arguments passed on to methods.
value	A character vector of the same length as attrs_order(x, ...).

Details

All classes in autodb contain an attrs_order attribute. It gives an easy way to find a list of all attributes/variables involved in an object, but its main purpose is to also assign those attributes a consistent order when printing or plotting the object.

Value

A character vector, giving attributes in the order in which they're prioritised for sorting within x.

---

autodb *Create a normalised database from a data frame*

Description

This is a wrapper function for applying normalise, autoref, and decompose. This takes a data frame and converts it straight into a database, which is the main intended use case for the package.

Usage

```
autodb(
  df,
  digits = getOption("digits"),
  single_ref = FALSE,
  ensure_lossless = TRUE,
  remove_avoidable = FALSE,
  constants_name = "constants",
  progress = FALSE,
  progress_file = "",
  ...
)
```

Arguments

df	a data.frame, containing the data to be normalised.
digits	a positive integer, indicating how many significant digits are to be used for numeric and complex variables. This is used for both pre-formatting in discover, and for rounding the data before use in decompose, so that the data satisfies the resulting schema. A value of NA results in no rounding. By default, this uses getOption("digits"), similarly to format. See the "Floating-point variables" section for discover for why this rounding is necessary for consistent results across different machines. See the note in print.default about digits >= 16.
single_ref	a logical, FALSE by default. If TRUE, then only one reference between each relation pair is kept when generating foreign key references. If a pair has multiple references, the kept reference refers to the earliest key for the child relation, as sorted by priority order.

`ensure_lossless`
: a logical, indicating whether to check whether the normalisation is lossless. If it is not, then an additional relation is added to the final "database", containing a key for `df`. This is enough to make the normalisation lossless.

`remove_avoidable`
: a logical, indicating whether to remove avoidable attributes in relations. If so, then an attribute are removed from relations if the keys can be changed such that it is not needed to preserve the given functional dependencies.

`constants_name`
: a scalar character, giving the name for any relation created to store constant attributes. If this is the same as a generated relation name, it will be changed, with a warning, to ensure that all relations have a unique name.

`progress`
: a logical, for whether to display progress to the user during dependency search in `discover`.

`progress_file`
: a scalar character or a connection. If `progress` is non-zero, determines where the progress is written to, in the same way as the `file` argument for `cat`.

`...`
: further arguments passed on to `discover`.

Details

Since `decompose` only works with functional dependencies, not approximate dependencies, the accuracy in `discover` is fixed as 1.

Value

A `database`, containing the data in `df` within the inferred database schema.

Examples

```
# simple example
autodb(ChickWeight)
```

---

autoref *Add foreign key references to a normalised database*

Description

Adds foreign key references to a `relation_schema` object automatically, replacing any existing references.

Usage

```
autoref(schema, single_ref = FALSE)
```

Arguments

schema	a relation_schema object, as given by synthesise.
single_ref	a logical, FALSE by default. If TRUE, then only one reference between each relation pair is kept when generating foreign key references. If a pair has multiple references, the kept reference refers to the earliest key for the child relation, as sorted by priority order.

Details

The method for generating references is simple. First, it finds every link between two relation schemas, where the parent contains all the attributes in one of the child's keys. This can be done separately for all of the child's keys, so there can be multiple links with the same parent and child if single_ref is TRUE.

Second, any transitive references are removed: if there are link relation pairs a -> b, b -> c, and a -> c, then the latter is transitive, and so is removed. If there is a cyclic reference, e.g. where c -> a, then the choice of which link to remove is arbitrary. Cycles cannot occur in sets of relation schemas resulting from decomposing a single table.

Value

A database_schema object, containing the given relation schemas and the created foreign key references.

Examples

```
rs <- relation_schema(
  list(
    a_b_c = list(c("a", "b", "c", "d"), list(c("a", "b", "c"))),
    a_b = list(c("a", "b", "d"), list(c("a", "b"), c("b", "d")))
  ),
  letters[1:4]
)
autoref(rs, single_ref = FALSE)
autoref(rs, single_ref = TRUE)
```

---

create *Create instance of a schema*

Description

Create a relation data object, using the given relational schema object, with the resulting relations empty and ready for data insertion using insert.

Usage

```
create(x, ...)
```

Arguments

x	a relational schema object, representing the schema to create an instance of, such as a relation_schema or database_schema object.
...	further arguments passed on to methods.

Value

An instance of the schema. For example, calling create on a database_schema creates a database, where all the relations contain zero records.

---

database *Databases*

Description

Enhances a relation object with foreign key reference information.

Usage

```
database(relations, references)
```

Arguments

relations	a relation object.
references	a list of references, each represented by a list containing four character elements. In order, the elements are a scalar giving the name of the child (referrer) schema, a vector giving the child attribute names, a scalar giving the name of the parent (referee) schema, and a vector giving the parent attribute names. The vectors must be of the same length and contain names for attributes present in their respective schemas, and the parent attributes must form a key.

Details

Unlike relation_schema and relation, and like database_schema, database is not designed to be vector-like: it only holds a single database. This adheres to the usual package use case, where a single data frame is being analysed at a time. However, it inherits from relation, so is vectorised with respect to its relations.

As with relation, duplicate relations, after ordering by attribute, are allowed, and can be removed with unique.

References, i.e. foreign key references, are allowed to have different attribute names in the child and parent relations; this can't occur in the output for autoref and normalise.

Subsetting removes any references that involve removed relations. Removing duplicates with unique changes references involving duplicates to involve the kept equivalent relations instead. Renaming relations with names<- also changes their names in the references.

Value

A database object, containing relations with references stored in an attribute of the same name. References are stored with their attributes in the order they appear in their respective relations.

Examples

```
rels <- relation(
  list(
    a = list(
      df = data.frame(a = logical(), b = logical()),
      keys = list("a")
    ),
    b = list(
      df = data.frame(b = logical(), c = logical()),
      keys = list("b", "c")
    )
  ),
  attrs_order = c("a", "b", "c", "d")
)
db <- database(
  rels,
  list(list("a", "b", "b", "b"))
)
print(db)
attrs(db)
stopifnot(identical(
  attrs(db),
  lapply(records(db), names)
))
keys(db)
attrs_order(db)
names(db)
references(db)

# relations can't reference themselves
## Not run:
  database(
    relation(
      list(a = list(df = data.frame(a = 1:5), keys = list("a"))),
      c("a", "b")
    ),
    list(list("a", "a", "a", "a"))
  )
  database(
    relation(
      list(a = list(df = data.frame(a = 1:5, b = 6:10), keys = list("a"))),
      c("a", "b")
    ),
    list(list("a", "b", "a", "a"))
  )
```

```
## End(Not run)

# an example with references between differently-named attributes
print(database(
  relation(
    list(
    citation = list(df = data.frame(citer = 1:5, citee = 6:10), keys = list(c("citer", "citee"))),
      article = list(df = data.frame(article = 1:10), keys = list("article"))
    ),
    c("citer", "citee", "article")
  ),
  list(
    list("citation", "citer", "article", "article"),
    list("citation", "citee", "article", "article")
  )
))

# inserting data
insert(db, data.frame(a = 1L, b = 2L, c = 3L, d = 4L))
# data is only inserted into relations where all columns are given...
insert(db, data.frame(a = 1L, b = 2L, c = 3L))
# and that are listed in relations argument
insert(
  db,
  data.frame(a = 1L, b = 2L, c = 3L, d = 4L),
  relations = "b"
)
# inserted data can't violate keys
## Not run:
  insert(
    db,
    data.frame(a = 1L, b = 1:2)
  )

## End(Not run)
# inserted data can't violate foreign key references
## Not run:
  insert(
    db,
    data.frame(a = 1L, b = 2L, c = 3L, d = 4L),
    relations = "a"
  )

## End(Not run)

# vector operations
db2 <- database(
  relation(
    list(
      e = list(df = data.frame(a = 1:5, e = 6:10), keys = list("e"))
    ),
    attrs_order = c("a", "e")
  ),
```

```
  list()
)
c(db, db2) # attrs_order attributes are merged
unique(c(db, db))

# subsetting
db[1]
stopifnot(identical(db[[1]], db[1]))
db[c(1, 2, 1, 2)] # replicates the foreign key references
c(db[c(1, 2)], db[c(1, 2)]) # doesn't reference between separate copies of db
unique(db[c(1, 2, 1, 2)]) # unique() also merges references

# another example of unique() merging references
db_merge <- database(
  relation(
    list(
      a = list(
        df = data.frame(a = logical(), b = logical()),
        keys = list("a")
      ),
      b = list(
        df = data.frame(b = logical(), c = logical(), d = logical()),
        keys = list("b")
      ),
      c_d = list(
        df = data.frame(c = logical(), d = logical(), e = logical()),
        keys = list(c("c", "d"))
      ),
      a.1 = list(
        df = data.frame(a = logical(), b = logical()),
        keys = list("a")
      ),
      b.1 = list(
        df = data.frame(b = logical(), c = logical(), d = logical()),
        keys = list("b")
      )
    ),
    c("a", "b", "c", "d", "e")
  ),
  list(
    list("a", "b", "b", "b"),
    list("b.1", c("c", "d"), "c_d", c("c", "d"))
  )
)
print(db_merge)
unique(db_merge)

# reassignment
# can't change keys included in references
## Not run: keys(db)[[2]] <- list("c")
# can't remove attributes included in keys
## Not run: attrs(db)[[2]] <- list("c", "d")
# can't remove attributes included in references
```

```
## Not run: attrs(db)[[1]] <- c("a", "d")
db3 <- db
# can change subset of schema, but loses references between altered and
# non-altered subsets
db3[2] <- database(
  relation(
    list(d = list(df = data.frame(d = logical(), c = logical()), keys = list("d"))),
    attrs_order(db3)
  ),
  list()
)
print(db3) # note the schema's name doesn't change
# names(db3)[2] <- "d" # this would change the name
keys(db3)[[2]] <- list(character()) # removing keys first...
# for a database_schema, we could then change the attrs for
# the second database. For a created relation, this is not
# allowed.
## Not run:
  attrs(db3)[[2]] <- c("b", "c")
  names(records(db3)[[2]]) <- c("b", "c")

## End(Not run)

# changing appearance priority for attributes
attrs_order(db3) <- c("d", "c", "b", "a")
print(db3)

# changing relation schema names changes them in references
names(db3) <- paste0(names(db3), "_long")
print(db3)

# reconstructing from components
db_recon <- database(
  relation(
    Map(list, df = records(db), keys = keys(db)),
    attrs_order(db)
  ),
  references(db)
)
stopifnot(identical(db_recon, db))
db_recon2 <- database(
  subrelations(db),
  references(db)
)
stopifnot(identical(db_recon2, db))

# can be a data frame column
data.frame(id = 1:2, relation = db)
```

---

database_schema *Database schemas*

Description

Enhances a relation_schema object with foreign key reference information.

Usage

```
database_schema(relation_schemas, references)
```

Arguments

relation_schemas	a relation_schema object, as returned by synthesise or relation_schema.
references	a list of references, each represented by a list containing four character elements. In order, the elements are a scalar giving the name of the child (referrer) schema, a vector giving the child attribute names, a scalar giving the name of the parent (referee) schema, and a vector giving the parent attribute names. The vectors must be of the same length and contain names for attributes present in their respective schemas, and the parent attributes must form a key.

Details

Unlike functional_dependency and relation_schema, database_schema is not designed to be vector-like: it only holds a single database schema. This adheres to the usual package use case, where a single data frame is being analysed at a time. However, it inherits from relation_schema, so is vectorised with respect to its relation schemas.

As with relation_schema, duplicate relation schemas, after ordering by attribute, are allowed, and can be removed with unique.

References, i.e. foreign key references, are allowed to have different attribute names in the child and parent relations; this can't occur in the output for autoref and normalise.

Subsetting removes any references that involve removed relation schemas. Removing duplicates with unique changes references involving duplicates to involve the kept equivalent schemas instead. Renaming relation schemas with names<- also changes their names in the references.

Value

A database_schema object, containing relation_schemas with references stored in an attribute of the same name. References are stored with their attributes in the order they appear in their respective relation schemas.

See Also

attrs, keys, attrs_order, and references for extracting parts of the information in a database_schema; create for creating a database object that uses the given schema; gv for converting the schema into Graphviz code; rename_attrs for renaming the attributes in attrs_order; reduce for filtering a schema's relations to those connected to a given relation by foreign key references; subschemas to return the relation_schema that the given schema contains; merge_empty_keys for combining relations with an empty key; merge_schemas for combining relations with matching sets of keys.

Examples

```
rs <- relation_schema(
  list(
    a = list(c("a", "b"), list("a")),
    b = list(c("b", "c"), list("b", "c"))
  ),
  attrs_order = c("a", "b", "c", "d")
)
ds <- database_schema(
  rs,
  list(list("a", "b", "b", "b"))
)
print(ds)
attrs(ds)
keys(ds)
attrs_order(ds)
names(ds)
references(ds)

# relations can't reference themselves
## Not run:
  database_schema(
    relation_schema(
      list(a = list("a", list("a"))),
      c("a", "b")
    ),
    list(list("a", "a", "a", "a"))
  )
  database_schema(
    relation_schema(
      list(a = list(c("a", "b"), list("a"))),
      c("a", "b")
    ),
    list(list("a", "b", "a", "a"))
  )

## End(Not run)

# an example with references between differently-named attributes
print(database_schema(
  relation_schema(
    list(
      citation = list(c("citer", "citee"), list(c("citer", "citee"))),
      article = list("article", list("article"))
    ),
    c("citer", "citee", "article")
  ),
  list(
    list("citation", "citer", "article", "article"),
    list("citation", "citee", "article", "article")
  )
))
```

```
# vector operations
ds2 <- database_schema(
  relation_schema(
    list(
      e = list(c("a", "e"), list("e"))
    ),
    attrs_order = c("a", "e")
  ),
  list()
)
c(ds, ds2) # attrs_order attributes are merged
unique(c(ds, ds))

# subsetting
ds[1]
stopifnot(identical(ds[[1]], ds[1]))
ds[c(1, 2, 1, 2)] # replicates the foreign key references
c(ds[c(1, 2)], ds[c(1, 2)]) # doesn't reference between separate copies of ds
unique(ds[c(1, 2, 1, 2)]) # unique() also merges references

# another example of unique() merging references
ds_merge <- database_schema(
  relation_schema(
    list(
      a = list(c("a", "b"), list("a")),
      b = list(c("b", "c", "d"), list("b")),
      c_d = list(c("c", "d", "e"), list(c("c", "d"))),
      a.1 = list(c("a", "b"), list("a")),
      b.1 = list(c("b", "c", "d"), list("b"))
    ),
    c("a", "b", "c", "d", "e")
  ),
  list(
    list("a", "b", "b", "b"),
    list("b.1", c("c", "d"), "c_d", c("c", "d"))
  )
)
print(ds_merge)
unique(ds_merge)

# reassignment
# can't change keys included in references
## Not run: keys(ds)[[2]] <- list("c")
# can't remove attributes included in keys
## Not run: attrs(ds)[[2]] <- list("c", "d")
# can't remove attributes included in references
## Not run: attrs(ds)[[1]] <- c("a", "d")
ds3 <- ds
# can change subset of schema, but loses references between altered and
# non-altered subsets
ds3[2] <- database_schema(
  relation_schema(
```

```
    list(d = list(c("d", "c"), list("d"))),
    attrs_order(ds3)
  ),
  list()
)
print(ds3) # note the schema's name doesn't change
# names(ds3)[2] <- "d" # this would change the name
keys(ds3)[[2]] <- list(character()) # removing keys first...
attrs(ds3)[[2]] <- c("b", "c") # so we can change the attrs legally
keys(ds3)[[2]] <- list("b", "c") # add the new keys
# add the reference lost during subset replacement
references(ds3) <- c(references(ds3), list(list("a", "b", "b", "b")))
stopifnot(identical(ds3, ds))

# changing appearance priority for attributes
attrs_order(ds3) <- c("d", "c", "b", "a")
print(ds3)

# changing relation schema names changes them in references
names(ds3) <- paste0(names(ds3), "_long")
print(ds3)

# reconstructing from components
ds_recon <- database_schema(
  relation_schema(
    Map(list, attrs(ds), keys(ds)),
    attrs_order(ds)
  ),
  references(ds)
)
stopifnot(identical(ds_recon, ds))
ds_recon2 <- database_schema(
  subschemas(ds),
  references(ds)
)
stopifnot(identical(ds_recon2, ds))

# can be a data frame column
data.frame(id = 1:2, schema = ds)
```

`decompose` *Decompose a data frame based on given normalised dependencies*

Description

Decomposes a data frame into several relations, based on the given database schema. It's intended that the data frame satisfies all the functional dependencies implied by the schema, such as if the schema was constructed from the same data frame. If this is not the case, the function will returns an error.

Usage

```
decompose(df, schema)
```

Arguments

df	a data.frame, containing the data to be normalised.
schema	a database schema with foreign key references, such as given by autoref.

Details

If the schema was constructed using approximate dependencies for the same data frame, decompose returns an error, to prevent either duplicate records or lossy decompositions. This is temporary: for the next update, we plan to add an option to allow this, or to add "approximate" equivalents of databases and database schemas.

Value

A database object, containing the data in df within the database schema given in schema.

---

dependant *Dependants*

Description

Generic function, with the only given method fetching dependants for functional dependencies.

Usage

```
dependant(x, ...)

dependant(x, ...) <- value
```

Arguments

x	an R object. For the given method, a functional_dependency.
...	further arguments passed on to methods.
value	A character vector of the same length as dependant(x, ...).

Value

A character vector containing dependants.

detset *Determinant sets*

Description

Generic function, with the only given method fetching determinant sets for functional dependencies.

Usage

```
detset(x, ...)

detset(x, ...) <- value
```

Arguments

x	an R object. For the given method, a functional_dependency.
...	further arguments passed on to methods.
value	A character vector of the same length as detset(x, ...).

Value

A list containing determinant sets, each consisting of a character vector with unique elements.

df_duplicated *Determine Duplicate Elements*

Description

duplicated "determines which elements of a vector or data frame are duplicates of elements with smaller subscripts, and returns a logical vector indicating which elements (rows) are duplicates". However, as of R 4.1, calling this on a data frame with zero columns always returns an empty logical vector. This has repercussions on other functions that use duplicated, such as unique and anyDuplicated. These functions add zero-column data frames as a special case.

Usage

```
df_duplicated(x, incomparables = FALSE, fromLast = FALSE, ...)

df_unique(x, incomparables = FALSE, fromLast = FALSE, ...)

df_anyDuplicated(x, incomparables = FALSE, fromLast = FALSE, ...)

df_records(x, use_rownames = FALSE, use_colnames = FALSE)
```

Arguments

x	a data frame.
incomparables	a vector of values that cannot be compared. FALSE is a special value, meaning that all values can be compared, and may be the only value accepted for methods other than the default. It will be coerced internally to the same type as x.
fromLast	logical indicating if duplication should be considered from the reverse side, i.e., the last (or rightmost) of identical elements would correspond to duplicated = FALSE.
...	arguments for particular methods.
use_rownames	a logical, FALSE by default, indicating whether row values should keep the row names from x. Defaults to FALSE.
use_colnames	a logical, FALSE by default, indicating whether row values should keep the column names from x for their elements. Defaults to FALSE.

Value

For df_duplicated, a logical vector with one element for each row.

For df_unique, a data frame is returned with the same columns, but possible fewer rows (and with row names from the first occurrences of the unique rows).

For df_anyDuplicated, an integer or real vector of length one with value the 1-based index of the first duplicate if any, otherwise 0.

For df_records, a list of the row values in x. This is based on a step in duplicated.data.frame. However, for data frames with zero columns, special handling returns a list of empty row values, one for each row in x. Without special handling, this step returns an empty list. This was the cause for duplicated returning incorrect results for zero-column data frames in older versions of R.

See Also

df_rbind

Examples

```
# row values for a 5x0 data frame
x <- data.frame(a = 1:5)[, FALSE, drop = FALSE]
do.call(Map, unname(c(list, x))) # original step returns empty list
df_records(x) # corrected version preserves row count
```

---

df_equiv *Test data frames for equivalence under row reordering*

Description

A convenience function, mostly used to testing that rejoin works as intended. It checks that data frames have the same dimensions and column names, with duplicates allowed, then checks they contain the same data. For the latter step, column names are made unique first, so columns with duplicate names must be presented in the same order in both data frames.

Usage

```
df_equiv(df1, df2, digits = getOption("digits"))
```

Arguments

df1, df2	Data frames.
digits	a positive integer, indicating how many significant digits are to be used for numeric and complex variables. A value of NA results in no rounding. By default, this uses getOption("digits"), similarly to format. See the note in print.default about digits >= 16.

Value

A logical.

---

df_rbind *Combine R Objects by Rows or Columns*

Description

rbind takes "a sequence of vector, matrix or data-frame arguments", and combines by rows for the latter. However, as of R 4.1, calling this on data frame with zero columns always returns zero rows, due to the issue mentioned for df_duplicated. This function adds zero-column data frames as a special case.

Usage

```
df_rbind(...)
```

Arguments

...	data frames.

Value

A data frame containing the ... arguments row-wise.

See Also

df_duplicated

discover *Dependency discovery with DFD*

Description

Finds all the minimal functional dependencies represented in a data frame.

Usage

```
discover(
  df,
  accuracy,
  digits = getOption("digits"),
  full_cache = TRUE,
  store_cache = TRUE,
  skip_bijections = FALSE,
  exclude = character(),
  exclude_class = character(),
  dependants = names(df),
  detset_limit = ncol(df) - 1L,
  progress = FALSE,
  progress_file = ""
)
```

Arguments

df	a data.frame, the relation to evaluate.
accuracy	a numeric in (0, 1]: the accuracy threshold required in order to conclude a dependency.
digits	a positive integer, indicating how many significant digits are to be used for numeric and complex variables. A value of NA results in no rounding. By default, this uses getOption("digits"), similarly to format. See the "Floating-point variables" section below for why this rounding is necessary for consistent results across different machines. See the note in print.default about digits >= 16.
full_cache	a logical, indicating whether to store information about how sets of attributes group the relation records (stripped partitions). Otherwise, only the number of groups is stored. Storing the stripped partition is expected to let the algorithm run more quickly, but might be inefficient for small data frames or small amounts of memory.
store_cache	a logical, indicating whether to keep cached information to use when finding dependencies for other dependants. This allows the algorithm to run more quickly by not having to re-calculate information, but takes up more memory.
skip_bijections	a logical, indicating whether to skip some dependency searches that are made redundant by discovered bijections between attributes. This can significantly

speed up the search if df contains equivalent attributes early in column order, but results in undefined behaviour if accuracy < 1. See Details for more information.

exclude — a character vector, containing names of attributes to not consider as members of determinant sets. If names are given that aren't present in df, the user is given a warning.

exclude_class — a character vector, indicating classes of attributes to not consider as members of determinant_sets. Attributes are excluded if they inherit from any given class.

dependants — a character vector, containing names of all attributes for which to find minimal functional dependencies for which they are the dependant. By default, this is all of the attribute names. A smaller set of attribute names reduces the amount of searching required, so can reduce the computation time if only some potential dependencies are of interest.

detset_limit — an integer, indicating the largest determinant set size that should be searched for. By default, this is large enough to allow all possible determinant sets. See Details for comments about the effect on the result, and on the computation time.

progress — a logical, for whether to display progress to the user during dependency search in discover.

progress_file — a scalar character or a connection. If progress is non-zero, determines where the progress is written to, in the same way as the file argument for cat.

Details

Column names for df must be unique.

The algorithm used for finding dependencies is DFD. This searches for determinant sets for each dependent attribute (dependant) by traversing the powerset of the other (non-excluded) attributes, and is equivalent to depth-first.

The implementation for DFD differs a little from the algorithm presented in the original paper:

- Some attributes, or attribute types, can be designated, ahead of time, as not being candidate members for determinant sets. This reduces the number of candidate determinant sets to be searched, saving time by not searching for determinant sets that the user would remove later anyway.
- Attributes that have a single unique value, i.e. are constant, get attributed a single empty determinant set. In the standard DFD algorithm, they would be assigned all the other non-excluded attributes as length-one determinant sets. Assigning them the empty set distinguishes them as constant, allowing for special treatment at normalisation and later steps.
- As was done in the original Python library, there is an extra case in seed generation for when there are no discovered maximal non-dependencies. In this case, we take all of the single-attribute nodes, then filter out by minimal dependencies as usual. This is equivalent to taking the empty set as the single maximal non-dependency.
- There are three results when checking whether a candidate node is minimal/maximal. TRUE indicates the node is minimal/maximal, as usual. FALSE has been split into FALSE and NA. NA indicates that we can not yet determine whether the node is minimal/maximal. FALSE indicates that we have determined that it is not minimal/maximal, and we can set its category as such. This is done by checking whether any of its adjacent subsets/supersets are

dependencies/non-dependencies, instead of waiting to exhaust the adjacent subsets/supersets to visit when picking the next node to visit.

- We do not yet keep hashmaps to manage subset/superset relationships, as described in Section 3.5 of the original paper.
- `skip_bijections` allows for additional optimisation for finding functional dependencies when there are pairwise-equivalent attributes.
- Missing values (NA) are treated as a normal value, with NA = NA being true, and x = NA being false for any non-NA value of x.

Floating-point variables:

Numerical/complex values, i.e. floating-point values, represent difficulties for stating functional dependencies. A fundamental condition for stating functional dependencies is that we can compare two values for the same variable, and they are equivalent or not equivalent.

Usually, this is done by checking they're equal – this is the approach used in `discover` – but we can use any comparison that is an equivalence relation.

However, checking floating-point values for equality is not simple. `==` is not appropriate, even when comparing non-calculated values we've read from a file, because how a given number is converted into a float can vary by computer architecture, meaning that two values can be considered equal on one computer, and not equal on another. This can happen even if they're both using 64-bit R, and even though all R platforms work with values conforming to the same standard (see `double`). For example, 8.54917750000000076227 and 8.54917749999999898591 are converted into different floating-point representations on x86, but the same representation on ARM, resulting in inequality and equality respectively.

For this and other reasons, checking numerical/complex values for (near-)equality in R is usually done with `all.equal`. This determines values x and y to be equal if their absolute/relative absolute difference is within some tolerance value. However, we can not use this. Equivalence relations must be transitive: if we have values x, y, and z, and x is equivalent to both y and z, then y and z must also be equivalent. This tolerance-based equivalence is not transitive: it is reasonably straightforward to set up three values so that the outer values are far enough apart to be considered non-equivalent, but the middle value is close enough to be considered equivalent to both of them. Using this to determine functional dependencies, therefore, could easily result in a large number of inconsistencies.

This means we have no good option for comparing numerical/complex values as-is for equivalence, with consistent results across different machines, so we must treat them differently. We have three options:

- Round/truncate the values, before comparison, to some low degree of precision;
- Coerce the values to another class before passing them into `discover`;
- Read values as characters if reading data from a file.

`discover` takes the first option, with a default number of significant digits low enough to ensure consistency across different machines. However, the user can also use any of these options when processing the data before passing it to `discover`. The third option, in particular, is recommended if reading data from a file.

Skipping bijections:

Skipping bijections allows skipping redundant searches. For example, if the search discovers that `A -> B` and `B -> A`, then only one of those attributes is considered for the remainder of the search.

Since the search time increases exponentially with the number of attributes considered, this can significantly speed up search times. At the moment, this is only be done for bijections between single attributes, such as `A <-> B`; if `A <-> {B, C}`, nothing is skipped. Whether bijections are skipped doesn't affect which functional dependencies are present in the output, but it might affect their order.

Skipping bijections for approximate dependencies, i.e. when `accuracy < 1`, should be avoided: it can result in incorrect output, since an approximate bijection doesn't imply equivalent approximate dependencies.

Limiting the determinant set size:

Setting `detset_limit` smaller than the largest-possible value has different behaviour for different search algorithms, the result is always that `discover(x, 1, detset_limit = n)` is equivalent to doing a full search, `fds <- discover(x, 1)`, then filtering by determinant set size post-hoc, `fds[lengths(detset(fds)) <= n]`.

For DFD, the naive way to implement it is by removing determinant sets larger than the limit from the search tree for possible functional dependencies for each dependant. However, this usually results in the search taking much more time than without a limit.

For example, suppose we search for determinant sets for a dependant that has none (the dependant is the only key for `df`, for example). Using DFD, we begin with a single attribute, then add other attributes one-by-one, since every set gives a non-dependency. When we reach a maximum-size set, we can mark all subsets as also being non-dependencies.

With the default limit, there is only one maximum-size set, containing all of the available attributes. If there are n candidate attributes for determinants, the search finishes after visiting n sets.

With a smaller limit k, there are $\binom{n}{k}$ maximum-size sets to explore. Since a DFD search adds or removes one attribute at each step, this means the search must take at least $k - 2 + 2\binom{n}{k}$ steps, which is larger than n for all non-trivial cases $0 < k \leq n$.

We therefore use a different approach, where any determinant sets above the size limit are not allowed to be candidate seeds for new search paths, and any discovered dependencies with a size above the limit are discard at the end of the entire DFD search. This means that nodes for determinant sets above the size limit are only visited in order to determine maximality of non-dependencies within the size limit. It turns out to be rare that this results in a significant speed-up, but it never results in the search having to visit more nodes than it would without a size limit, so the average search time is never made worse.

Value

A `functional_dependency` object, containing the discovered dependencies. The column names of `df` are stored in the `attrs` attribute, in order, to serve as a default priority order for the attributes during normalisation.

References

Abedjan Z., Schulze P., Naumann F. (2014) DFD: efficient functional dependency discovery. *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management (CIKM '14). New York, U.S.A.*, 949–958.

Examples

```
# simple example
discover(ChickWeight, 1)

# example with spurious dependencies
discover(CO2, 1)
# exclude attributes that can't be determinants.
# in this case, the numeric attributes are now
# not determined by anything, because of repeat measurements
# with no variable to mark them as such.
discover(CO2, 1, exclude_class = "numeric")
# include only dependencies with dependants of interest.
discover(CO2, 1, dependants = c("Treatment", "uptake"))
```

`functional_dependency` *Functional dependency vectors*

Description

Creates a set of functional dependencies with length-one dependants.

Usage

```
functional_dependency(FDs, attrs_order, unique = TRUE)
```

Arguments

`FDs`	a list of functional dependencies, in the form of two-elements lists: the first element contains a character vector of all attributes in the determinant set, and the second element contains the single dependent attribute (dependant).
`attrs_order`	a character vector, giving the names of all attributes. These need not be present in `FDs`, but all attributes in `FDs` must be present in `attrs`.
`unique`	a logical, TRUE by default, for whether to remove duplicate dependencies.

Details

When several sets of functional dependencies are concatenated, their `attrs_order` attributes are merged, so as to preserve all of the original attribute orders, if possible. If this is not possible, because the orderings disagree, then the returned value of the `attrs_order` attribute is their union instead.

Value

A `functional_dependency` object, containing the list given in `FDs`, with `attrs_order` an attribute of the same name. Functional dependencies are returned with their determinant sets sorted according to the attribute order in `attrs`. Any duplicates found after sorting are removed.

See Also

detset, dependant, and attrs_order for extracting parts of the information in a functional_dependency; rename_attrs for renaming the attributes in attrs_order.

Examples

```
fds <- functional_dependency(
  list(list(c("a", "b"), "c"), list(character(), "d")),
  attrs_order = c("a", "b", "c", "d")
)
print(fds)
detset(fds)
dependant(fds)
attrs_order(fds)

# vector operations
fds2 <- functional_dependency(list(list("e", "a")), c("a", "e"))
c(fds, fds2) # attrs_order attributes are merged
unique(c(fds, fds))

# subsetting
fds[1]
fds[c(1, 2, 1)]
stopifnot(identical(fds[[2]], fds[2]))

# reassignment
fds3 <- fds
fds3[2] <- functional_dependency(list(list("a", "c")), attrs_order(fds3))
print(fds3)
detset(fds3)[[2]] <- character()
dependant(fds3)[[2]] <- "d"
stopifnot(identical(fds3, fds))
# changing appearance priority for attributes
attrs_order(fds3) <- rev(attrs_order(fds3))
fds3

# reconstructing from components
fds_recon <- functional_dependency(
 Map(list, detset(fds), dependant(fds)),
 attrs_order(fds)
)
stopifnot(identical(fds_recon, fds))

# can be a data frame column
data.frame(id = 1:2, fd = fds)

# (in)equality ignores header
stopifnot(all(fds3 == fds))
stopifnot(!any(fds != fds))
```

gv *Generate Graphviz input text to plot objects*

Description

Produces text input for Graphviz to make an HTML diagram of a given object.

Usage

```
gv(x, name = NA_character_, ...)
```

Arguments

x	an object to be plotted.
name	a scalar character, giving the name of the object, if any. This name is used for the resulting graph, to allow for easier combining of graphs into a single diagram if required.
...	further arguments passed to or from other methods.

Details

Details of what is plotted are given in individual methods. There are expected commonalities, which are described below.

The object is expected to be one of the following:

- an object whose elements have the same length. Examples would be data frames, matrices, and other objects that can represent relations, with names for the elements, and an optional name for the object itself.
- a graph of sub-objects, each of which represent a relation as described above, possibly with connections between the objects, and an optional name for the graph as a whole.

Each relation is presented as a record-like shape, with the following elements:

- A optional header with the relation's name, and the number of (unique) records.
- A set of rows, one for each attribute in the relation. These rows have the following contents:
  - the attribute names.
  - a depiction of the relation's (candidate) keys. Each column represents a key, and a filled cell indicates that the attribute in that row is in that key. The keys are given in lexical order, with precedence given to keys with fewer attributes, and keys with attributes that appear earlier in the original data frame's attribute order. Default output from other package functions will thus have the primary key given first. In the future, this will be changed to always give the primary key first.
  - optionally, the attribute types: specifically, the first element when passing the attribute's values into class.

Any foreign key references between relations are represented by one-way arrows, one per attribute in the foreign key.

If the object has a name, this name is attached to the resulting graph in Graphviz. This is to allow easier combination of several such graphs into a single image, if a user wishes to do so.

Value

A scalar character, containing text input for Graphviz.

See Also

gv.data.frame, gv.relation_schema, gv.database_schema, gv.relation, and gv.database for individual methods.

Examples

```
# simple data.frame example
txt_df <- gv(ChickWeight, "chick")
cat(txt_df)
if (requireNamespace("DiagrammeR", quietly = TRUE)) {
  DiagrammeR::grViz(txt_df)
}
# simple database example
db <- autodb(ChickWeight)
txt_db <- gv(db)
cat(txt_db)
if (requireNamespace("DiagrammeR", quietly = TRUE)) {
  DiagrammeR::grViz(txt_db)
}
# simple relation schemas
rschema <- synthesise(discover(ChickWeight, 1))
txt_rschema <- gv(rschema)
cat(txt_rschema)
if (requireNamespace("DiagrammeR", quietly = TRUE)) {
  DiagrammeR::grViz(txt_rschema)
}
# simple database schema
dschema <- normalise(discover(ChickWeight, 1))
txt_dschema <- gv(dschema)
cat(txt_dschema)
DiagrammeR::grViz(txt_dschema)
# simple relations
rel <- create(synthesise(discover(ChickWeight, 1)))
txt_rel <- gv(rel)
cat(txt_rel)
if (requireNamespace("DiagrammeR", quietly = TRUE)) {
  DiagrammeR::grViz(txt_rel)
}
```

gv.data.frame *Generate Graphviz input text to plot a data frame*

Description

Produces text input for Graphviz to make an HTML diagram of a given data frame.

Usage

```
## S3 method for class 'data.frame'
gv(x, name = NA_character_, ...)
```

Arguments

x	a data.frame.
name	a character scalar, giving the name of the record, if any. The name must be non-empty, since it is also used to name the single table in the plot. Defaults to NA: if left missing, it is set to "data".
...	further arguments passed to or from other methods.

Details

The rows in the plotted data frame include information about the attribute classes.

Value

A scalar character, containing text input for Graphviz.

See Also

The generic gv.

gv.database *Generate Graphviz input text to plot databases*

Description

Produces text input for Graphviz to make an HTML diagram of a given database.

Usage

```
## S3 method for class 'database'
gv(x, name = NA_character_, ...)
```

Arguments

x	a database, as returned by autoref or autodb.
name	a scalar character, giving the name of the database, if any. This name is used for the resulting graph, to allow for easier combining of graphs into a single diagram if required.
...	further arguments passed to or from other methods.

Details

Each relation in the database is presented as a set of rows, one for each attribute in the relation. These rows include information about the attribute classes.

Value

A scalar character, containing text input for Graphviz.

See Also

The generic gv.

---

gv.database_schema *Generate Graphviz input text to plot database schemas*

Description

Produces text input for Graphviz to make an HTML diagram of a given database schema.

Usage

```
## S3 method for class 'database_schema'
gv(x, name = NA_character_, ...)
```

Arguments

x	a database schema, as given by normalise, synthesise, or autoref.
name	a character scalar, giving the name of the schema, if any.
...	further arguments passed to or from other methods.

Details

Each relation in the schema is presented as a set of rows, one for each attribute in the relation. These rows do not include information about the attribute classes.

Any foreign key references are represented by arrows between the attribute pairs.

Value

A scalar character, containing text input for Graphviz.

See Also

The generic gv.

`gv.relation` *Generate Graphviz input text to plot relations*

Description

Produces text input for Graphviz to make an HTML diagram of a given relation.

Usage

```
## S3 method for class 'relation'
gv(x, name = NA_character_, ...)
```

Arguments

x	a relation.
name	a character scalar, giving the name of the schema, if any.
...	further arguments passed to or from other methods.

Details

Each relation is presented as a set of rows, one for each attribute in the relation. These rows include information about the attribute classes.

Value

A scalar character, containing text input for Graphviz.

See Also

The generic gv.

gv.relation_schema *Generate Graphviz input text to plot relation schemas*

Description

Produces text input for Graphviz to make an HTML diagram of a given relation schema.

Usage

```
## S3 method for class 'relation_schema'
gv(x, name = NA_character_, ...)
```

Arguments

x	a relation schema, as given by relation_schema or synthesise.
name	a character scalar, giving the name of the schema, if any.
...	further arguments passed to or from other methods.

Details

Each relation in the schema is presented as a set of rows, one for each attribute in the relation. These rows do not include information about the attribute classes.

Value

A scalar character, containing text input for Graphviz.

See Also

The generic gv.

insert *Insert data*

Description

Generic function for inserting a data frame of data into an object.

Usage

```
insert(x, vals, relations = names(x), all = FALSE, ...)
```

Arguments

x	a relational data object, into which to insert data, such as a relation or database object.
vals	a data frame, containing data to insert.
relations	a character vector, containing names of elements of x into which to insert data. By default, insert attempts to insert data into every element.
all	a logical, indicating whether vals is required to contain all attributes of all elements of x[relations]. By default, it is not, and data is only inserted into elements of x[relations] whose attributes are all present in vals.
...	further arguments pass on to methods.

Details

This function is intended for inserting into an object that is itself comprised of data frames, such as a relation or a database. The given methods have the following behaviour:

- If an empty set of data is inserted, into a non-empty object element, nothing happens.
- If an empty set of data is inserted into an empty object element, the resulting element is also empty, but takes on the attribute/column classes of the inserted data. This is done to prevent having to know attribute classes during object creation.
- Insertion can fail if inserting would violate object constraints. For example, databases cannot have data inserted that would violate candidate/foreign key constraints.
- For other cases, the data is inserted in an object element in the same way as using rbind, followed by unique.

While key violations prevent insertion, re-insertion of existing records in an object element does not. This makes insertion equivalent to an INSERT OR IGNORE expression in SQL. In particular, it is somewhat like using this expression in SQLite, since that implementation uses dynamic typing.

If vals contains attributes not included in attrs_order(x), insert throws an error, since those attributes can't be inserted.

If a partial set of attributes is inserted, and all is FALSE, then data is only inserted into components of x[relations] whose required attributes are all present in vals. If all is TRUE, insert returns an error instead. This is useful when specifying relations: in that case, you often intend to insert into all of the specified elements, so not including all the required attributes is a mistake, and all = TRUE prevents it.

If all is TRUE, insert throws an error in this case: This ensures you insert into all members of a specified value of relations.

Value

An R object of the same class as x, containing the additional new data.

keys *Relational data keys*

Description

Generic function, with the only given method fetching candidate key lists for relation schemas.

Usage

```
keys(x, ...)

keys(x, ...) <- value
```

Arguments

x	a relational schema object, such as a relation_schema or database_schema object, or a relational data object, such as a relation or database object.
...	further arguments passed on to methods.
value	A list of lists of character vectors, of the same length as keys(x, ...). The number of keys for an element of x can be changed.

Value

A list containing lists of unique character vectors, representing candidate keys for each element of x.

merge_empty_keys *Merge relation schemas with empty keys*

Description

Merges an object's schemas with empty keys. The remaining such schema contains all attributes contained in such schemas.

Usage

```
merge_empty_keys(x)
```

Arguments

x	a relational schema object, such as a relation_schema or database_schema object.

Details

This function is not itself generic, but makes use of the generic functions keys and merge_schemas. Any input class with valid methods for these generic functions can be passed into this function.

For database_schema objects, references involving the schemas with empty keys are updated to refer to the merged schema.

Value

An R object of the same class as x, where relations with an empty key have been merged into a single relation.

See Also

merge_schemas, on which this function is based.

---

merge_schemas *Merge relation schemas in given pairs*

Description

Generic function that merges pairs of an object's schemas with matching sets of keys. The remaining schemas contain all the attributes from the schemas merged into them.

Usage

```
merge_schemas(x, to_remove, merge_into, ...)
```

Arguments

x	a relational schema object, such as a relation_schema or database_schema object.
to_remove	an integer vector, giving the indices for schemas to be merged into other schemas, then removed.
merge_into	an integer vector of the same length as to_remove, giving the indices for the schemas into which to merge.
...	further arguments passed on to methods.

Value

An R object of the same class as x, where the relations have been merged as indicated.

See Also

merge_empty_keys, which is based on this function.

Examples

```
rs <- relation_schema(
  list(
    a = list(c("a", "b"), list("a")),
    b = list(c("b", "c"), list("b")),
    b.1 = list(c("b", "d"), list("b")),
    d = list(c("d", "e"), list("d", "e"))
  ),
  letters[1:5]
)
ds <- database_schema(
  rs,
  list(
    list("a", "b", "b", "b"),
    list("b.1", "d", "d", "d")
   )
)
merge_schemas(rs, 3, 2) # merging b and b.1
merge_schemas(ds, 3, 2) # also merging their references

# merging a schema into itself just removes it
merge_schemas(rs, 3, 3)
merge_schemas(ds, 3, 3)
```

`normalise` *Create normalised database schemas from functional dependencies*

Description

Creates a database schema from given functional dependencies, satisfying at least third normal form, using Bernstein's synthesis.

Usage

```
normalise(
  dependencies,
  single_ref = FALSE,
  ensure_lossless = TRUE,
  reduce_attributes = TRUE,
  remove_avoidable = FALSE,
  constants_name = "constants",
  progress = FALSE,
  progress_file = ""
)
```

Arguments

`dependencies` a `functional_dependency` object, as given by `discover`.

`single_ref` a logical, FALSE by default. If TRUE, then only one reference between each relation pair is kept when generating foreign key references. If a pair has multiple references, the kept reference refers to the earliest key for the child relation, as sorted by priority order.

`ensure_lossless` a logical, TRUE by default. If TRUE, and the decomposition isn't lossless, an extra relation is added to make the decomposition lossless.

`reduce_attributes` a logical, TRUE by default. If TRUE, `dependencies` are checked for determinant attributes that are made redundant by the other dependencies. This is redundant if `dependencies` is output from `discover`, since there will be no such redundant attributes.

`remove_avoidable` a logical, indicating whether to remove avoidable attributes in relations. If so, then an attribute are removed from relations if the keys can be changed such that it is not needed to preserve the given functional dependencies.

`constants_name` a scalar character, giving the name for any relation created to store constant attributes. If this is the same as a generated relation name, it will be changed, with a warning, to ensure that all relations have a unique name.

`progress` a logical, for whether to display progress to the user during dependency search in `discover`.

`progress_file` a scalar character or a connection. If `progress` is non-zero, determines where the progress is written to, in the same way as the `file` argument for `cat`.

Details

This is a wrapper function for applying `synthesise` and `autoref`, in order. For creating relation schemas and foreign key references separately, use these functions directly. See both functions for examples.

For details on the synthesis algorithm used, see `synthesise`.

Value

A `database_schema` object, containing the synthesis relation schemas and the created foreign key references.

---

nudge *Nudge meta-analysis data*

Description

Data used for a meta-analysis on the effectiveness of nudges, i.e. choice architecture interventions.

Usage

```
nudge
```

Format

A data frame with 447 effect size measurements and 25 columns:

1. publication_id, integer ID number for the publication. Note that two publications were erroneously assigned the same ID number, so this is not a unique publication identifier.
2. study_id, integer ID number for the study.
3. es_id, integer ID number for the effect size measured.
4. reference, publication citation in "Author(s) (year)" format. Due to two publications being assigned the same reference, this is also not a unique publication identifier.
5. title, title of the publication. Due to the error in assigning publication ID numbers, this is the unique publication identifier within the data set.
6. year, year of the publication.
7. location, geographical location of the intervention. This is given as a factor, rather than an integer, using the information provided in the codebook.
8. domain, factor giving the intervention's behavioural domain.
9. intervention_category, factor giving the intervention's category, based on the taxonomy in Münscher et al. (2016).
10. intervention_technique, factor giving the intervention's technique, based on the taxonomy in Münscher et al. (2016).
11. type_experiment, factor giving the type of experiment, as defined by Harrison and List (2004).
12. population, factor giving the intervention's target population. This is given as a factor, rather than an integer, using the information provided in the codebook.
13. n_study, sample size of the overall study.
14. n_comparison, combined sample size of the control and the intervention for the measured effect size.
15. n_control, sample size of the control condition for the measured effect size.
16. n_intervention, sample size of the intervention condition for the measured effect size.
17. binary_outcome, logical for whether the outcome scale is binary or continuous.
18. mean_control, mean of outcome for the control condition.
19. sd_control, SD of outcome for the control condition.
20. mean_intervention, mean of outcome for the intervention condition.
21. sd_intervention, SD of outcome for the intervention condition.
22. cohens_d, extracted effect size of intervention.
23. variance_d, variance of extracted effect size.
24. approximation, logical for whether effect size extraction involved approximation.
25. wansink, logical for whether the study was (co-)authored by Brian Wansink. This was added on revision, because, a few years before publication, Wansink had many papers retracted or corrected, due to various questionable practices, resulting in Wansink being determined to have committed scientific misconduct. This column was added to check whether the findings were robust to the exclusion of non-retracted studies by the Cornell Food and Brand Laboratory, of which Wansink was the director.

Source

https://osf.io/fywae/

References

Mertens S., Herberz M., Hahnel U. J. J., Brosch T. (2022) The effectiveness of nudging: A meta-analysis of choice architecture interventions across behavioral domains. *Proc. Natl. Acad. Sci. U.S.A.*, **4**, 119(1).

---

records *Relational data records*

Description

Generic function, for retrieving data contained in a database-like structure. In particular, this is intended for such structures where the individual relations can't be accessed with subsetting.

Usage

```
records(x, ...)

records(x, ...) <- value
```

Arguments

x	a relational data object, such as a relation or database object.
...	further arguments passed on to methods.
value	A list of data frames of the same length as records(x, ...), where each data frame has the same column names as that which it will replace, in the same order.

Details

Since the relational data objects in autodb, relation and database, have subsetting methods that return relational data objects, the data contained within them can't be accessed by subsetting. This function is intended for accessing it instead.

It's recommended to call records before doing any subsetting, since subsetting on a relation data object does more work that will be thrown away, such as subsetting on a database checking whether foreign key references should be removed.

Value

A list containing data frames, with elements named for their respective relations.

Examples

```
db <- autodb(ChickWeight)
records(db) # data for Chick and Time_Chick relations

# ways to get data for subsets
records(db)[c(1, 2)]
records(db)[[1]]
records(db)$Chick

# subsetting first isn't recommended: removes foreign key
# reference as mentions, and you need to subset again anyway
records(db[[1]])[[1]]
```

`reduce` *Remove relations not linked to the main relations*

Description

Filters an object's relations, keeping only the main relations, and those considered ancestors via foreign key references. Foreign key references involving removed relations are also removed.

Usage

```
reduce(x, ...)
```

Arguments

`x`	An object whose relations are to be filtered.
`...`	further arguments passed to or from other methods.

Details

Details on how the main tables are chosen are given in individual methods.

This function is mostly intended for simplifying a database, or a database schema, for the purposes of exploration, particularly by examining plots. While the filtering might remove important auxiliary relations, it's also likely to remove any based on spurious dependencies, of which some databases can contain many.

Value

An object of the same class as `x`, with the auxiliary relations and foreign key references removed.

See Also

`reduce.database_schema`, `reduce.database`.

reduce.database *Remove database relations not linked to the main relations*

Description

Filters a database's relations, keeping only the main relations, and those considered ancestors via foreign key references. Foreign key references involving removed relations are also removed.

Usage

```
## S3 method for class 'database'
reduce(x, ...)
```

Arguments

x	A database, whose relations are to be filtered.
...	further arguments passed to or from other methods.

Details

The main relations are considered to be the relations with the largest number of records.

Using rejoin on the database resulting from reduce is likely to fail or return incomplete results.

Value

A database, with the auxiliary relations and foreign key references removed.

reduce.database_schema *Remove database schema relations not linked to the given relations*

Description

Filters a database schema's relations, keeping only the given relations, and those considered ancestors via foreign key references. Foreign key references involving removed relations are also removed.

Usage

```
## S3 method for class 'database_schema'
reduce(x, main, ...)
```

Arguments

x	A database schema, whose relations are to be filtered.
main	A character vector, containing names of relations to be considered as the "main" relations.
...	further arguments passed to or from other methods.

Details

This method takes a given set of main relations, rather than inferring them.

Using rejoin on the database resulting from decomposing a data frame with the reduced schema is likely to fail or return incomplete results.

Value

A database schema, with the auxiliary relations and foreign key references removed.

---

references *Schema references*

Description

Generic function, returning present (foreign key) references.

Usage

```
references(x, ...)

references(x) <- value
```

Arguments

x	an R object with references, such as a database_schema or database object.
...	further arguments passed on to methods.
value	A list, of the same length as references(x, ...).

Value

A list, giving references.

`rejoin` *Join a database into a data frame*

Description

Rejoins the relations in a database into a single data frame, if possible. This is the inverse of calling `autodb`, except that the rows might be returned in a different order.

Usage

```
rejoin(database)
```

Arguments

`database`	A database containing the data to be rejoined, as returned by `decompose`.

Details

The rejoining algorithm might not use all of the given relations: it begins with the relation with the largest number of records, then joins it with enough relations to contain all of the present attributes. This is not limited to relations that the starting relation is linked to by foreign keys, and is not limited to them either, since in some cases this constraint would make it impossible to rejoin with all of the present attributes.

Since the algorithm may not use all of the given relations, the algorithm may ignore some types of database inconsistency, where different relations hold data inconsistent with each other. In this case, the rejoining will be lossy. Rejoining the results of `reduce` can also be lossy.

Due to the above issues, the algorithm will be changed to use all of the relations in the future.

Not all databases can be represented as a single data frame. A simple example is any database where the same attribute name is used for several difference sources of data, since rejoining results in inappropriate merges.

Value

A data frame, containing all information contained `database` if it is lossless and self-consistent.

Examples

```
# simple example
db <- autodb(ChickWeight)
rj <- rejoin(db)
rj <- rj[order(as.integer(rownames(rj))), ]
all(rj == ChickWeight) # TRUE

# showing rejoin() doesn't check for inconsistency:
# add another Chick table with the diets swapped
db2 <- db[c(1, 2, 1)]
records(db2)[[3]]$Diet <- rev(records(db2)[[3]]$Diet)
```

```
rj2 <- rejoin(db2)
rj2 <- rj2[order(as.integer(rownames(rj2))), ]
all(rj2 == ChickWeight) # TRUE
```

`relation` *Relation vectors*

Description

Creates a set of relation schemas, including the relation's attributes and candidate keys.

Usage

```
relation(relations, attrs_order)
```

Arguments

`relations`	a named list of relations, in the form of two-element lists: the first element contains a data frame, where the column names are the attributes in the associated schema, and the second element contains a list of character vectors, each representing a candidate key.
`attrs_order`	a character vector, giving the names of all attributes. These need not be present in `schemas`, but all attributes in `schemas` must be present in `attrs_order`.

Details

Relation vectors are unlikely to be needed by the user directly, since they are essentially `database` objects that can't have foreign key references. They are mostly used to mirror the use of the vector-like `relation_schema` class for the `database_schema` class to be a wrapper around. This makes creating a `database` from a `relation_schema` a two-step process, where the two steps can be done in either order: creation with `create` and `insert`, and adding references with `database_schema` or `database`.

Duplicate schemas, after ordering by attribute, are allowed, and can be removed with `unique`.

When several sets of relation schemas are concatenated, their `attrs_order` attributes are merged, so as to preserve all of the original attribute orders, if possible. If this is not possible, because the orderings disagree, then the returned value of the `attrs_order` attribute is their union instead.

Value

A `relation` object, containing the list given in `relations`, with `attrs_order` stored in an attribute of the same name. Relation schemas are returned with their keys' attributes sorted according to the attribute order in `attrs_order`, and the keys then sorted by priority order. Attributes in the data frame are also sorted, first by order of appearance in the sorted keys, then by order in `attrs_order` for non-prime attributes.

See Also

records, attrs, keys, and attrs_order for extracting parts of the information in a relation_schema; gv for converting the schema into Graphviz code; rename_attrs for renaming the attributes in attrs_order.

Examples

```
rels <- relation(
  list(
    a = list(
      df = data.frame(a = logical(), b = logical()),
      keys = list("a")
    ),
    b = list(
      df = data.frame(b = logical(), c = logical()),
      keys = list("b", "c")
    )
  ),
  attrs_order = c("a", "b", "c", "d")
)
print(rels)
records(rels)
attrs(rels)
stopifnot(identical(
  attrs(rels),
  lapply(records(rels), names)
))
keys(rels)
attrs_order(rels)
names(rels)

# inserting data
insert(rels, data.frame(a = 1L, b = 2L, c = 3L, d = 4L))
# data is only inserted into relations where all columns are given...
insert(rels, data.frame(a = 1L, b = 2L, c = 3L))
# and that are listed in relations argument
insert(
  rels,
  data.frame(a = 1L, b = 2L, c = 3L, d = 4L),
  relations = "a"
)

# vector operations
rels2 <- relation(
  list(
    e = list(
      df = data.frame(a = logical(), e = logical()),
      keys = list("e")
    )
  ),
  attrs_order = c("a", "e")
)
```

```
c(rels, rels2) # attrs_order attributes are merged
unique(c(rels, rels))

# subsetting
rels[1]
rels[c(1, 2, 1)]
stopifnot(identical(rels[[1]], rels[1]))

# reassignment
rels3 <- rels
rels3[2] <- relation(
  list(
    d = list(
      df = data.frame(d = logical(), c = logical()),
      keys = list("d")
    )
  ),
  attrs_order(rels3)
)
print(rels3) # note the relation's name doesn't change
# names(rels3)[2] <- "d" # this would change the name
keys(rels3)[[2]] <- list(character()) # removing keys first...
# for a relation_schema, we could then change the attrs for
# the second relation. For a created relation, this is not
# allowed.
## Not run:
  attrs(rels3)[[2]] <- c("b", "c")
  names(records(rels3)[[2]]) <- c("b", "c")

## End(Not run)

# changing appearance priority for attributes
rels4 <- rels
attrs_order(rels4) <- c("d", "c", "b", "a")
print(rels4)

# reconstructing from components
rels_recon <- relation(
  Map(list, df = records(rels), keys = keys(rels)),
  attrs_order(rels)
)
stopifnot(identical(rels_recon, rels))

# can be a data frame column
data.frame(id = 1:2, relation = rels)
```

`relation_schema` *Relation schema vectors*

Description

Creates a set of relation schemas, including the relation's attributes and candidate keys.

Usage

```
relation_schema(schemas, attrs_order)
```

Arguments

schemas	a named list of schemas, in the form of two-element lists: the first element contains a character vector of all attributes in the relation schema, and the second element contains a list of character vectors, each representing a candidate key.
attrs_order	a character vector, giving the names of all attributes. These need not be present in schemas, but all attributes in schemas must be present in attrs_order.

Details

Duplicate schemas, after ordering by attribute, are allowed, and can be removed with \code{\link{unique}}.

When several sets of relation schemas are concatenated, their attrs_order attributes are merged, so as to preserve all of the original attribute orders, if possible. If this is not possible, because the orderings disagree, then the returned value of the attrs_order attribute is their union instead.

Value

A relation_schema object, containing the list given in schemas, with attrs_order stored in an attribute of the same name. Relation schemas are returned with their keys' attributes sorted according to the attribute order in attrs_order, and the keys then sorted by priority order. Attributes in the schema are also sorted, first by order of appearance in the sorted keys, then by order in attrs_order for non-prime attributes.

See Also

attrs, keys, and attrs_order for extracting parts of the information in a relation_schema; create for creating a relation object that uses the given schema; gv for converting the schema into Graphviz code; rename_attrs for renaming the attributes in attrs_order; merge_empty_keys for combining relations with an empty key; merge_schemas for combining relations with matching sets of keys.

Examples

```
schemas <- relation_schema(
  list(
    a = list(c("a", "b"), list("a")),
    b = list(c("b", "c"), list("b", "c"))
  ),
  attrs_order = c("a", "b", "c", "d")
)
print(schemas)
attrs(schemas)
keys(schemas)
attrs_order(schemas)
names(schemas)

# vector operations
```

```
schemas2 <- relation_schema(
  list(
    e = list(c("a", "e"), list("e"))
  ),
  attrs_order = c("a", "e")
)
c(schemas, schemas2) # attrs_order attributes are merged
unique(c(schemas, schemas))

# subsetting
schemas[1]
schemas[c(1, 2, 1)]
stopifnot(identical(schemas[[1]], schemas[1]))

# reassignment
schemas3 <- schemas
schemas3[2] <- relation_schema(
  list(d = list(c("d", "c"), list("d"))),
  attrs_order(schemas3)
)
print(schemas3) # note the schema's name doesn't change
# names(schemas3)[2] <- "d" # this would change the name
keys(schemas3)[[2]] <- list(character()) # removing keys first...
attrs(schemas3)[[2]] <- c("b", "c") # so we can change the attrs legally
keys(schemas3)[[2]] <- list("b", "c") # add the new keys
stopifnot(identical(schemas3, schemas))

# changing appearance priority for attributes
attrs_order(schemas3) <- c("d", "c", "b", "a")
print(schemas3)

# reconstructing from components
schemas_recon <- relation_schema(
  Map(list, attrs(schemas), keys(schemas)),
  attrs_order(schemas)
)
stopifnot(identical(schemas_recon, schemas))

# can be a data frame column
data.frame(id = 1:2, schema = schemas)
```

`rename_attrs` *Rename relational data attributes*

Description

Generic function, for renaming attributes present in a database-like structure.

Usage

```
rename_attrs(x, names, ...)
```

Arguments

x	an object with an attrs_order attribute. This includes relational schema objects, such as a relation_schema or database_schema object, relational data objects, such as a relation or database object, and functional_dependency objects.
names	a character vector of the same length as attrs_order(x), with no duplicated elements, to be used as the new attribute names.
...	further arguments passed on to methods.

Details

This function has a different intended use to re-assigning attrs_order: that is intended only for rearranging the order of the attributes, without renaming them. This is intended for renaming the attributes without re-ordering them.

Value

A relational object of the same type as x, with attributes renamed consistently across the whole object.

---

subrelations *Database subrelations*

Description

Generic function, returning subrelations for x.

Usage

```
subrelations(x, ...)
```

Arguments

x	an R object, intended to be some sort of database-like object that contains relations, such as a database object.
...	further arguments passed on to methods.

Value

A relation-type object, or a list of relation-type objects if the subrelation isn't vectorised. For example, if x is a database, the result is the contained relation.

subschemas *Schema subschemas*

Description

Generic function, returning subschemas for `x`.

Usage

```
subschemas(x, ...)
```

Arguments

`x`	an R object, intended to be some sort of schema that contains other schemas, such as a `database_schema` object.
`...`	further arguments passed on to methods.

Value

A schema-type object, or a list of schema-type objects if the subschema isn't vectorised. For example, if `x` is a `database_schema`, the result is the contained `relation_schema`.

synthesise *Synthesise relation schemas from functional dependencies*

Description

Synthesises the dependency relationships in dependencies into a database schema satisfying at least third normal form, using Bernstein's synthesis.

Usage

```
synthesise(
  dependencies,
  ensure_lossless = TRUE,
  reduce_attributes = TRUE,
  remove_avoidable = FALSE,
  constants_name = "constants",
  progress = FALSE,
  progress_file = ""
)
```

Arguments

dependencies	a functional_dependency object, as given by discover.
ensure_lossless	a logical, TRUE by default. If TRUE, and the decomposition isn't lossless, an extra relation is added to make the decomposition lossless.
reduce_attributes	a logical, TRUE by default. If TRUE, dependencies are checked for determinant attributes that are made redundant by the other dependencies. This is redundant if dependencies is output from discover, since there will be no such redundant attributes.
remove_avoidable	a logical, indicating whether to remove avoidable attributes in relations. If so, then an attribute are removed from relations if the keys can be changed such that it is not needed to preserve the given functional dependencies.
constants_name	a scalar character, giving the name for any relation created to store constant attributes. If this is the same as a generated relation name, it will be changed, with a warning, to ensure that all relations have a unique name.
progress	a logical, for whether to display progress to the user during dependency search in discover.
progress_file	a scalar character or a connection. If progress is non-zero, determines where the progress is written to, in the same way as the file argument for cat.

Details

Bernstein's synthesis is a synthesis algorithm for normalisation of a set of dependencies into a set of relations that are in third normal form. This implementation is based on the version given in the referenced paper.

The implementation also includes a common additional step, to ensure that the resulting decomposition is lossless, i.e. a relation satisfying the given dependencies can be perfectly reconstructed from the relations given by the decomposition. This is done by adding an additional relation, containing a key for all the original attributes, if one is not already present.

As an additional optional step, schemas are checked for "avoidable" attributes, that can be removed without loss of information.

Constant attributes, i.e. those whose only determinant set is empty, get assigned to a relation with no keys.

Output is independent of the order of the input dependencies: schemas are sorted according to their simplest keys.

Schemas are sorted before ensuring for losslessness, or removing avoidable attributes. As a result, neither optional step changes the order of the schemas, and ensuring losslessness can only add an extra schema to the end of the output vector.

Value

A relation_schema object, containing the synthesised relation schemas.

References

3NF synthesis algorithm: Bernstein P. A. (1976) Synthesizing third normal form relations from functional dependencies. *ACM Trans. Database Syst.*, **1, 4**, 277–298.

Removal of avoidable attributes: Ling T., Tompa F. W., Kameda T. (1981) An improved third normal form for relational databases. *ACM Trans. Database Syst.*, **6, 2**, 329–346.

Examples

```
# example 6.24 from The Theory of Relational Databases by David Maier
# A <-> B, AC -> D, AC -> E, BD -> C
deps <- functional_dependency(
  list(
    list("A", "B"),
    list("B", "A"),
    list(c("A", "C"), "D"),
    list(c("A", "C"), "E"),
    list(c("B", "D"), "C")
  ),
  attrs_order = c("A", "B", "C", "D", "E")
)
synthesise(deps, remove_avoidable = FALSE)
synthesise(deps, remove_avoidable = TRUE)
```

Index