# Package: powerjoin (via r-universe)

September 14, 2024

**Title** Extensions of 'dplyr' and 'fuzzyjoin' Join Functions

**Version** 0.1.0

**Description** We extend 'dplyr' and 'fuzzyjoin' join functions with features to preprocess the data, apply various data checks, and deal with conflicting columns.

**License** MIT + file LICENSE

**Encoding** UTF-8

**LazyData** true

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.3.2.9000

**Imports** dplyr, glue, rlang, tidyselect, vctrs, purrr, tibble, tidyr, cli, methods

**URL** https://github.com/moodymudskipper/powerjoin

**BugReports** https://github.com/moodymudskipper/powerjoin/issues

**Suggests** testthat (>= 3.0.0)

**Config/testthat/edition** 3

**Repository** https://moodymudskipper.r-universe.dev

**RemoteUrl** https://github.com/moodymudskipper/powerjoin

**RemoteRef** HEAD

**RemoteSha** e1cdbf2ce625c2e92c7540dbe9fd5fdb65ed36b5

## Contents

check_specs *Build a checklist for power joins*

## Description

Build a checklist for power joins

## Usage

```
check_specs(
  implicit_keys = c("inform", "ignore", "warn", "abort"),
  column_conflict = c("ignore", "inform", "warn", "abort"),
  duplicate_keys_left = c("ignore", "inform", "warn", "abort"),
  duplicate_keys_right = c("ignore", "inform", "warn", "abort"),
  unmatched_keys_left = c("ignore", "inform", "warn", "abort"),
  unmatched_keys_right = c("ignore", "inform", "warn", "abort"),
  missing_key_combination_left = c("ignore", "inform", "warn", "abort"),
  missing_key_combination_right = c("ignore", "inform", "warn", "abort"),
  inconsistent_factor_levels = c("ignore", "inform", "warn", "abort"),
  inconsistent_type = c("ignore", "inform", "warn", "abort"),
  grouped_input = c("ignore", "inform", "warn", "abort"),
  na_keys = c("ignore", "inform", "warn", "abort")
)
```

## Arguments

| | |
|---|---|
| implicit_keys | What to do if keys are not given explicitly through the by argument |
| column_conflict | What to do if the join creates a column conflict which is not handled by the conflict argument |
| duplicate_keys_left | What to do if we find duplicate sets of keys in the left table |
| duplicate_keys_right | What to do if we find duplicate sets of keys in the right table |
| unmatched_keys_left | What to do if we find unmatched sets of keys in the left table |
| unmatched_keys_right | What to do if we find unmatched sets of keys in the right table |
| missing_key_combination_left | What to do if the left table doesn't contain all key combinations |
| missing_key_combination_right | What to do if the right table doesn't contain all key combinations |
| inconsistent_factor_levels | What to do if the key columns from both sides have inconsistent factor levels |

| | |
|---|---|
| inconsistent_type | What to do if we joined keys have a different type |
| grouped_input | What to do if one or both of the tables are grouped |
| na_keys | What to do if keys contain missing values |

## Value

A character vector of class "powerjoin_check"

## Examples

```
check_specs(
  implicit_keys = "ignore",
  grouped_input = "inform",
  column_conflict = "abort",
  na_keys ="warn")
```

---

# coalesce_xy *Coalesce helpers*

---

## Description

These are wrappers around dplyr::coalesce, designed for convenient use in the conflict argument of **powerjoin**'s join functions. coalesce_xy() is just like dplyr::coalesce (except it takes only 2 arguments), coalesce_yx() looks first in y and then in x if y is missing.

## Usage

```
coalesce_xy(x, y)

coalesce_yx(x, y)
```

## Arguments

| | |
|---|---|
| x | A vector |
| y | A vector |

## Value

A vector

## Examples

```
coalesce_xy(c(NA, 2, 3), c(11, 12, NA))
coalesce_yx(c(NA, 2, 3), c(11, 12, NA))
```

## extended-equality *Extended equality operators*

### Description

`%==%` is the bone operator, it works like `==` but `NA %==% 1` is `FALSE` and `NA %==% NA` is `TRUE`. `%in.%` is the a vectorized `%in%`, that can be seen as a rowwise `%in%` when applied to data frame columns. These are convenient helpers for fuzzy joins.

### Usage

```
x %==% y

x %in.% y
```

### Arguments

| | |
|---|---|
| x | A vector |
| y | A vector for `%==%`, a list of vectors for `%in.%` |

### Examples

```
df1 <- data.frame(key = c("b", "z"))
df2 <- data.frame(key1 = c("a", "b", "c"), key2 = c("x", "y", "z"), val = 1:3)
power_left_join(df1, df2, ~ .x$key %in.% list(.y$key1, .y$key2))

df3 <- data.frame(key1 = c("a", NA))
df4 <- data.frame(key2 = c("a", "b", NA), val = 1:3)

# note the difference
power_inner_join(df3, df4, by = ~ .x$key1 == .y$key2)
power_inner_join(df3, df4, by = ~ .x$key1 %==% .y$key2)

# typically we would only use the conditions above as part of more complex conditions.
# In this precise case they are equivalent to these equi joins
power_inner_join(df3, df4, by = c(key1 = "key2"))
power_inner_join(df3, df4, by = c(key1 = "key2"), na_matches = "never")
```

## full_diagnostic *Inform on all potential issues*

### Description

This is the output of `check_specs()` with all arguments set to `"inform"`, it's useful for a complete join diagnostic.

## Usage

```
full_diagnostic
```

## Format

An object of class `powerjoin_check` of length 12.

---

`paste_xy` *Paste helpers*

---

## Description

These are similar to `paste()` but by default ignore `NA` and empty strings (`""`). If they are found in a conflicting column we return the value from the other column without using the separator. If both columns have such values we return an empty string.

## Usage

```
paste_xy(x, y, sep = " ", na = NULL, ignore_empty = TRUE)

paste_yx(x, y, sep = " ", na = NULL, ignore_empty = TRUE)
```

## Arguments

| | |
|---|---|
| `x` | A vector |
| `y` | A vector |
| `sep` | separator |
| `na` | How to treat `NA`s, they are ignored by default, if `NA` the result will be `NA`, just as with `stringr::str_c`, if `"NA"` NAs will be coerced to character just as with `paste()`. Any other string can be used |
| `ignore_empty` | Whether to ignore empty strings, to avoid trailing and leading separators |

## Value

A character vector

## Examples

```
paste_xy(letters[1:3], c("d", NA, ""))
paste_yx(letters[1:3], c("d", NA, ""))
paste_xy(letters[1:3], c("d", NA, ""), na = NA, ignore_empty = FALSE)
paste_xy(letters[1:3], c("d", NA, ""), na = "NA", ignore_empty = FALSE)
```

power_left_join *Power joins*

## Description

Power joins

## Usage

```
power_left_join(
  x,
  y = NULL,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  keep = NULL,
  na_matches = c("na", "never"),
  check = check_specs(),
  conflict = NULL,
  fill = NULL
)

power_right_join(
  x,
  y = NULL,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  keep = NULL,
  na_matches = c("na", "never"),
  check = check_specs(),
  conflict = NULL,
  fill = NULL
)

power_inner_join(
  x,
  y = NULL,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  keep = NULL,
  na_matches = c("na", "never"),
  check = check_specs(),
  conflict = NULL,
  fill = NULL
)
```

```
power_full_join(
  x,
  y = NULL,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  keep = NULL,
  na_matches = c("na", "never"),
  check = check_specs(),
  conflict = NULL,
  fill = NULL
)
```

## Arguments

| | |
|---|---|
| x, y | A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details. |
| by | As in **dplyr**, but extended so user can supply a formula or a list of character and formulas. Formulas are used for fuzzy joins, see dedicated section below. |
| copy | Ignored at the moment because **powerjoin** doesn't support databases |
| suffix | If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2. |
| keep | A boolean for compatibility with **dplyr**, or a value among "left", "right", "both", "none" or "default". See dedicated section below. |
| na_matches | Should two NA or two NaN values match?<br>• "na", the default, treats two NA or two NaN values as equal, like %in%, match(), and merge().<br>• "never" treats two NA or two NaN values as different, and will never match them together or to any other values. This is similar to joins for database sources and to base::merge(incomparables = NA). |
| check | A list created with check_specs() |
| conflict | A function, formula, the special value amongst "patch", or a named list of such items. If the LHS of the formula is rw the rhs will be applied rowwise. Note that the columns will be subsetted with [ so for list columns .x or .y will refer to length 1 lists and you might sometimes need .x[[1]] or .y[[1]]. |
| fill | Values used to replace missing values originating in unmatched keys, or a named list of such items. |

## Value

A data frame

### keep argument values

- NULL (default) : merge keys and name them as the left table's keys, and keep columns used for fuzzy joins from both tables

- `left` : keep only key columns for left table
- `right`: keep only key columns for right table
- `both` or `TRUE`: keep key columns from both tables, adding suffix if relevant
- `none` : drop all key columns from the output
- `FALSE` : merge keys and name them as the left table's keys, maps to `none` for fuzzy joins

### fuzzy joins

To specify fuzzy matching conditions we use formulas in which the we refer to the columns from the left side data frame using `.x` and the right side data frame using `.y`, for instance `by = ~ .x$col1 > .y$col2`.

We can specify several condition and even mix equi condition with fuzzy condition, for instance `by = c(col1 = "col2", ~ .x$col3 > .y$col4)`

To fuzzy match strings we can leverage the functions from the **stringr** package since they are vectorized on all main arguments, for instance to match observations where `col1` contains `col1` we can attach **stringr** and do `by = ~ str_detect(.x$col1, fixed(.y$col2))`.

Another useful function is `stringdist` from the **stringdist** package to match strings that are close enough, for instance `by = ~ stringdist::stringdist(.x$a,.y$a) < 2`

We can also define a new column computed during the fuzzy matching, using the arrow assignment operator, for instance : `by = ~ .x$col1 > (mysum <- .y$col2 + .y$col3)`

When the by condition evaluates to `NA` the observation is dismissed. This makes `by = c(a = "b")` slightly different from `by = ~ .x$a == .y$b` when `na_matches` is `"na"` (the default). To be able to match `NA` with `NA` in fuzzy matching condition we can use the `%==%` operator (bone operator), defined in this package.

### Examples

```
# See README for a more verbose version
library(tibble)
male_penguins <- tribble(
  ~name,    ~species,     ~island, ~flipper_length_mm, ~body_mass_g,
  "Giordan",    "Gentoo",    "Biscoe",               222L,        5250L,
  "Lynden",    "Adelie", "Torgersen",              190L,        3900L,
  "Reiner",    "Adelie",     "Dream",              185L,        3650L
)

female_penguins <- tribble(
  ~name,    ~species,  ~island, ~flipper_length_mm, ~body_mass_g,
  "Alonda",    "Gentoo", "Biscoe",              211,        4500L,
  "Ola",    "Adelie",  "Dream",             190,        3600L,
  "Mishayla",    "Gentoo", "Biscoe",              215,        4750L,
)

# apply different checks
power_inner_join(
  male_penguins[c("species", "island")],
  female_penguins[c("species", "island")],
  check = check_specs(implicit_keys = "ignore", duplicate_keys_right = "inform")
```

```
)

df1 <- tibble(id = 1:3, value = c(10, NA, 30))
df2 <- tibble(id = 2:4, value = c(22, 32, 42))

# handle conflicted columns when joining
power_left_join(df1, df2, by = "id", conflict = `+`)

# the most frequent use case is to coalesce
power_left_join(df1, df2, by = "id", conflict = coalesce_xy)
power_left_join(df1, df2, by = "id", conflict = coalesce_yx)

# the conflict function is applied colwise by default!
power_left_join(df1, df2, by = "id", conflict = ~ sum(.x, .y, na.rm = TRUE))

# apply conflict function rowwise
power_left_join(df1, df2, by = "id", conflict = rw ~ sum(.x, .y, na.rm = TRUE))

# subset columns without repeating keys
power_inner_join(
  male_penguins %>% select_keys_and(name),
  female_penguins %>% select_keys_and(female_name = name),
  by = c("species", "island")
)

# semi join
power_inner_join(
  male_penguins,
  female_penguins %>% select_keys_and(),
  by = c("species", "island")
)

# agregate without repeating keys
power_left_join(
  male_penguins %>% summarize_by_keys(male_weight = mean(body_mass_g)),
  female_penguins %>% summarize_by_keys(female_weight = mean(body_mass_g)),
  by = c("species", "island")
)

# pack auxiliary colums without repeating keys
power_left_join(
  male_penguins %>% pack_along_keys(name = "m"),
  female_penguins %>% pack_along_keys(name = "f"),
  by = c("species", "island")
)

# fuzzy join
power_inner_join(
  male_penguins %>% select_keys_and(male_name = name),
  female_penguins %>% select_keys_and(female_name = name),
  by = c(~.x$flipper_length_mm < .y$flipper_length_mm, ~.x$body_mass_g > .y$body_mass_g)
)
```

```
# fuzzy + equi join
power_inner_join(
  male_penguins %>% select_keys_and(male_name = name),
  female_penguins %>% select_keys_and(female_name = name),
  by = c("island", ~.x$flipper_length_mm > .y$flipper_length_mm)
)

# define new column without repeating computation
power_inner_join(
  male_penguins %>% select_keys_and(male_name = name),
  female_penguins %>% select_keys_and(female_name = name),
  by = ~ (mass_ratio <- .y$body_mass_g / .x$body_mass_g) > 1.2
)
power_inner_join(
  male_penguins %>% select_keys_and(male_name = name),
  female_penguins %>% select_keys_and(female_name = name),
  by = ~ (mass_ratio <- .y$body_mass_g / .x$body_mass_g) > 1.2,
  keep = "none"
)

# fill unmatched values
df1 <- tibble(id = 1:3)
df2 <- tibble(id = 1:2, value2 = c(2, NA), value3 = c(NA, 3))
power_left_join(df1, df2, by = "id", fill = 0)
power_left_join(df1, df2, by = "id", fill = list(value2 = 0))

# join recursively
df1 <- tibble(id = 1, a = "foo")
df2 <- tibble(id = 1, b = "bar")
df3 <- tibble(id = 1, c = "baz")
power_left_join(list(df1, df2, df3), by = "id")
power_left_join(df1, list(df2, df3), by = "id")
```

---

`preprocess_inputs` *Preprocess powerjoin inputs*

---

## Description

These functions are named after the tidyverse functions `select`, `summarize`, `nest`, `pack`, `pivot_wider` and `pivot_longer` and are designed to avoid repetition of key columns when preprocessing the data for a join. They should only be used in the `x` and `y` arguments of **powerjoin** join functions. No further transformation should be applied on top of them.

## Usage

```
select_keys_and(.data, ...)

summarize_by_keys(.data, ...)
```

```
nest_by_keys(.data, ..., name = NULL)

pack_along_keys(.data, ..., name)

complete_keys(.data)
```

## Arguments

| | |
|---|---|
| .data | A data frame to pivot. |
| ... | Additional arguments passed on to methods. |
| name | Name of created column |

## Details

Unlike their tidyverse counterparts these just add an attribute to the input and don't reshape it. The join function then preprocesses the inputs using these attributes and the keys.

## Value

A data frame identical to the .data but with a "powerjoin_preprocess" attribute to be handled by the join functions

## Examples

```
# in practice you'll mostly use those in join function calls directly
x <- select_keys_and(head(iris, 2), Sepal.Width)
# all it does is add an attribute that will be processed by the join function
attr(x, "powerjoin_preprocess")
# see `?power_left_join` or README for practical examples
```

# Index