



---

# *Journal of Statistical Software*

MMMMMM YYYY, Volume VV, Issue II.

*doi: 10.18637/jss.v000.i00*

---

## **cubble: An R Package for Structuring Spatio-temporal Data**

**H. Sherry Zhang**  
Monash University

**Dianne Cook**  
Monash University

**Ursula Laa**  
University of Natural Resources and Life Sciences

**Nicolas Langrené**  
CSIRO Data61

**Patricia Menéndez**  
Monash University

---

### **Abstract**

The abstract of the article.

*Keywords:* spatio-temporal data, R.

---

# 1. Introduction

## Motivation

Many data structures have been proposed for spatial (**sf** by Pebesma (2018)) and temporal (**tsibble** by Wang, Cook, and Hyndman (2020)) data in the R community, while less has been done for spatio-temporal data. The lack of such tools could potentially because analysts usually treat the spatial and temporal dimension of the data separately, without realising the need to create a new data structure. While this approach follows the third tidy data principal (Wickham 2014) (*Each type of observational unit forms a table*), analysts always need to manually join results from different observational units or combining multiple tables into one for downstream analysis. This additional step doesn't add new operations into the data but can be error prone.

## Existing packages

Currently, available spatio-temporal data structure in R includes: **spacetime** (Pebesma 2012), which proposes four space-time layouts: Full grid (STF), sparse grid (STS), irregular (STI), and trajectory (STT). The data structure it uses is based on **sp** (Pebesma and Bivand 2005) and **xts** (Ryan and Ulrich 2020), both of which has been replaced by more recent implementations. **spatstat** (Baddeley and Turner 2005) implements a **ppp** class for point pattern data; and more recent, **stars** (Pebesma 2021) implements a spatio-temporal array with the dplyr's data cube structure **cubelyr** (Wickham 2020) as its backend. While these implementations either store spatial and temporal variables all in a single table, hence duplicate the spatial variables for each temporal unit; or split them into two separate tables that has the problem of manually joining, mentioned in the previously. None of these packages enjoy both the benefits of being able to separate manipulation in the two dimensions while also keep the data object as a whole. This create a gap in the software development. The requirement for such a tool is important given the ubiquity of spatio-temporal vector data in the wild: the Ireland wind data from **gstat** is an classic example data that splits variables into spatial (**wind.loc**) and temporal (**wind**) dimension; Bureau of Meteorology (BoM) provides climate observations that are widely applied in agriculture and ecology study; air pollution data.

**Our new data structure for spatio-temporal data** This paper describes the implementation of a new spatio-temporal data structure: **cubble**. **cubble** implements a relational data structure that uses two forms to manage the switch between spatial and temporal dimension. With this structure, users can manipulate the spatial or temporal dimension separately, while leaves the linking of two dimensions to `\pkg{cubble}\pkg{}`. The software is available from the Comprehensive R Archive Network (CRAN) at [CRAN link].

## Section division

The rest of the paper will be divided as follows: [complete when the paper structure is more solid]

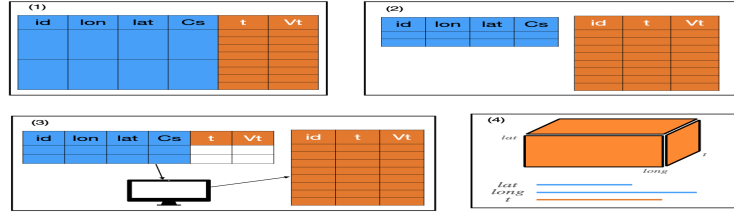


Figure 1: Illustration of incoming data formats for spatio-temporal data. (1) Data comes in as a single table; (2) Separate tables for spatial and temporal variables; (3) A single table with all the parameters used to query the database and a separate table for queried data; and (4) Cubical data in array or NetCDF format.

## 2. The cubble package

Spatio-temporal data usually come in various forms and Figure 1 shows four examples of this. No matter which form the data is in, these formats share some common components that characterise spatio-temporal data. A spatial identifier (`id` in the diagram) is the unique identifier of each site. The temporal identifier (`t` in the diagram) prescribes the time stamp each site is recorded. Coordinates, comprising of latitude and longitude (`lon` and `lat` in the diagram), locates each site on the map. These identifiers will be the building blocks for the data structure introduced below. Other variables in the data can be categorised into two groups: spatial variables that are invariant at each time stamp for every site, i.e. the name or code of the weather station and temporal variables that varies with time.

In a cubble, there are two forms: nested form and long form, and Figure 2 sketches the two forms along with the associated attributes. The decision on which form to use is output-oriented, meaning analysts need to first think about whether the output of a particular operation is identified only by the spatial identifier, or a combination of spatial and temporal identifier. The nested cubble is suitable for working with operations that are only identified by site and this type of operation can be a pure manipulation of spatial variables, or a summary of temporal variables by site (i.e. the output of counting the number of raining day is only identified by sites and hence should be performed with the nested form). Underneath the nested form, a cubble is built from a row-wise dataframe (`rowwise_df`) where each site forms a separate group. This structure simplifies the calculation that involves temporal variables by avoiding the use of `map` syntax when working with list-column.

For those operations whose output involves both a spatial and temporal dimension, long form should be used. The long form is identified by both the spatial and temporal identifier and adopts a grouped dataframe (`grouped_df`) to forms each site as a group. Spatial variables are stored separately in a **tibble** as an special attribute of the long cubble. This design avoids repeating the spatial variables at each time stamp while not dropping information from spatial variables.

### 2.1. Create a cubble in the nested form

To use functionalities from cubble, data analysts first need to create a cubble. `as_cubble` create a **cubble** by supplying the three key components: **key** as the spatial identifier; **index**

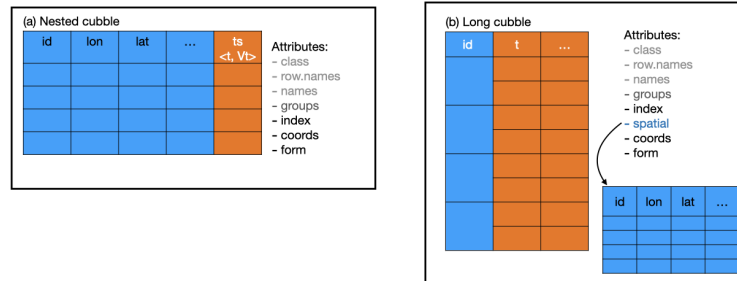


Figure 2: Illustration of nested and long cubble.

as the temporal identifier; and a vector of `coords` in the order of longitude first and then latitude. The naming of `key` and `index` follows the convention in the `tsibble` package. The cubble created by default is in the nested form. Below is an example of creating a cubble:

```
R> (cubble_nested <- cubble::climate_flat %>%
+   as_cubble(key = id, index = date, coords = c("long", "lat")))

# cubble:   id [5]: nested form
# bbox:     [115.97, -32.94, 133.55, -12.42]- check gap on long and lat
# temporal: date [date], prcp [dbl], tmax [dbl], tmin [dbl]
#   id      lat long elev name      wmo_id ts
#   <chr>    <dbl> <dbl> <dbl> <chr>    <dbl> <list>
1 ASN00009021 -31.9 116. 15.4 perth airport 94610 <tibble [366 x 4]>
2 ASN00010311 -31.9 117. 179  york          94623 <tibble [366 x 4]>
3 ASN00010614 -32.9 117. 338  narrogin    94627 <tibble [366 x 4]>
4 ASN00014015 -12.4 131. 30.4 darwin airport 94120 <tibble [366 x 4]>
5 ASN00015131 -17.6 134. 220  elliot     94236 <tibble [366 x 4]>
```

There are a few information in the **cubble** header: the name of the `key` variable, `bbox`, and also the name of variable nested in the `ts` column. In this example, each site is identifier is `id` and the number in the bracket means there are 5 unique `id` in this dataset. The `bbox` in the second row gives the range of the coordinates. The temporal variables are all nested in the `ts` column, but it could be useful to know the name these variables. The third row in the cubble header shows these names and in this example this includes: precipitation, `prcp`, maximum temperature, `tmax`, and minimum temperature, `tmin`.

## 2.2. Stretch a nested cubble into the long form

The long cubble is suitable to manipulate the time dimension of the data. The function `stretch()` switches the nested cubble into the long cubble by first extracts all the spatial variables into a separate tibble and store in the `spatial` attribute and then unnests the `ts` column:

```
R> (cubble_long <- cubble_nested %>% stretch(ts))

# cubble:  date, id [5]: long form
# bbox:    [115.97, -32.94, 133.55, -12.42]- check gap on long and lat
# spatial: lat [dbl], long [dbl], elev [dbl], name [chr], wmo_id [dbl]
   id      date      prcp  tmax  tmin
   <chr>    <date>    <dbl> <dbl> <dbl>
1 ASN00009021 2020-01-01      0  31.9  15.3
2 ASN00009021 2020-01-02      0  24.9  16.4
3 ASN00009021 2020-01-03      6  23.2  13
4 ASN00009021 2020-01-04      0  28.4  12.4
5 ASN00009021 2020-01-05      0  35.3  11.6
6 ASN00009021 2020-01-06      0  34.8  13.1
7 ASN00009021 2020-01-07      0  32.8  15.1
8 ASN00009021 2020-01-08      0  30.4  17.4
9 ASN00009021 2020-01-09      0  28.7  17.3
10 ASN00009021 2020-01-10      0  32.6  15.8
# ... with 1,820 more rows
```

Notice here that the third line in the header now shows the name of spatial variables rather than the temporal variables.

## 2.3. Tamp a long cubble back to the nested form

Manipulation on the spatial and temporal dimension can be an iterative process. Many times, analysts will need to go back and forth between the nested and long cubble. The `stretch()` function introduced in the previous section switches a nested cubble into a long cubble and function `tamp()` is its inverse function to switch a long cubble back to the nested cubble:

```
R> (cubble_back <- cubble_long %>% tamp())

# cubble:  id [5]: nested form
# bbox:    [115.97, -32.94, 133.55, -12.42]- check gap on long and lat
# temporal: date [date], prcp [dbl], tmax [dbl], tmin [dbl]
   id      lat  long  elev name      wmo_id ts
   <chr>    <dbl> <dbl> <dbl> <chr>    <dbl> <list>
1 ASN00009021 -31.9  116.  15.4 perth airport  94610 <tibble [366 x 4]>
2 ASN00010311 -31.9  117.  179  york      94623 <tibble [366 x 4]>
3 ASN00010614 -32.9  117.  338  narrogin  94627 <tibble [366 x 4]>
4 ASN00014015 -12.4  131.  30.4 darwin airport  94120 <tibble [366 x 4]>
5 ASN00015131 -17.6  134.  220  elliott   94236 <tibble [366 x 4]>
```

## 2.4. Migrate spatial variables to a long cubble

As a final data output for modelling or visualisation, spatio-temporal data is usually expected to be in a single table. Function `migrate()` moves the spatial variables from the `spatial` attribute into the long cubble:

```
R> (cubble_long %>% migrate(long, lat))

# cubble:  date, id [5]: long form
# bbox:    [115.97, -32.94, 133.55, -12.42]- check gap on long and lat
# spatial: lat [dbl], long [dbl], elev [dbl], name [chr], wmo_id [dbl]
  id      date      prcp tmax tmin long  lat
<chr>    <date>    <dbl> <dbl> <dbl> <dbl> <dbl>
1 ASN00009021 2020-01-01      0  31.9  15.3  116. -31.9
2 ASN00009021 2020-01-02      0  24.9  16.4  116. -31.9
3 ASN00009021 2020-01-03      6  23.2  13    116. -31.9
4 ASN00009021 2020-01-04      0  28.4  12.4  116. -31.9
5 ASN00009021 2020-01-05      0  35.3  11.6  116. -31.9
6 ASN00009021 2020-01-06      0  34.8  13.1  116. -31.9
7 ASN00009021 2020-01-07      0  32.8  15.1  116. -31.9
8 ASN00009021 2020-01-08      0  30.4  17.4  116. -31.9
9 ASN00009021 2020-01-09      0  28.7  17.3  116. -31.9
10 ASN00009021 2020-01-10      0  32.6  15.8  116. -31.9
# ... with 1,820 more rows
```

In this workflow described above, data objects come into cubble in the nested form, then various operations on the spatial and temporal dimension can go back and forth between the nested and long form, and finally, the data will come out of cubble in the long form for further modelling or visualisation.

Building from an underlying `tbl_df` structure, it is natural to implement methods available in `dplyr` to cubble. Supported methods in the cubble with `dplyr` generics includes:

basics	<b>mutate, filter, summarise, select, arrange, rename, left_join</b>
grouping	<b>group_by, ungroup</b>
slice family	<b>slice_head, slice_tail, slice_sample, slice_min and slice_max</b>

**cubble** is also compatible with **tsibble** in the sense that the original list-column can be a `tbl_ts` object. Duplicates and gaps should be first checked before structuring the data into a cubble. If the input data is a **tsibble** object, the long form cubble is also a **tsibble** where users can directly apply time series operations.

## 3. Advanced features/ considerations

### 3.1. Hierarchical structure

```
switch_key()
```

### 3.2. Data fusion and matching

### 3.3. Interactive graphics

Interactive graphics can listen to users' actions on the plot to provide additional information that facilitates data exploration. This is a useful technique for spatio-temporal data since users can zoom or pan the map to view the local and global structure of the map; use tooltips or popups to query more information about a graphic element; or highlight points to explore its linked views in other plots. In the R community, many implementations have been developed to connect R to `javascript` to create interactive graphics. In relation to spatio, temporal, and spatio-temporal data, the general purpose packages **plotly** (Sievert 2020) and **leaflet** (Cheng, Karambelkar, and Xie 2021) realise various interactive actions through their corresponding `javascript` libraries. **crosstalk** (Cheng and Sievert 2021) and **tsibbletalk** (Wang and Cook 2020) implement brushed linking between `htmlwidgets`. **ggiraph** (Gohel and Skintzos 2021) enables tooltip, self-linking, and customised actions specified through its own `javascript`.

While many graphic implementations present worked examples to illustrate the usage of the package, few documents the underlying pipeline that transforms the raw data step-by-step to the final view on the screen. There have been some early work in building the data pipeline for (interactive) graphics (Buja, Asimov, and Hurley 1988; Buja, Cook, and Swayne 1996; Sutherland, Rossini, Lumley, Lewin-Koh, Dickerson, Cox, and Cook 2000) and more recent discussions include Wickham, Lawrence, Cook, Buja, Hofmann, and Swayne (2009), Xie, Hofmann, and Cheng (2014), and Cheng, Cook, and Hofmann (2016).

Figure 3 shows how cross-linking works with the two forms in a cubble. The data pipeline flows from the *Augmented Data* to *Transformation*, and then *Subset*, to finally, *Plot*. This data pipeline, proposed by Wickham *et al.* (2009), places the *Subset* stage as the last step before *Plot* so that each plot can be made separately from different subsets. This design fits naturally with cubble where the spatial map is solely made from the nested form and the time series plot can be created from the long form. When a user action is captured either from the map or the time series, the relevant row will be highlighted in the data and all the observations that shares the same `key` variable (`id`), in both forms, will then be highlighted. This type of linking is called categorical linking (Xie *et al.* 2014), which is a one-to-many linking through highlighting observations within the same category.

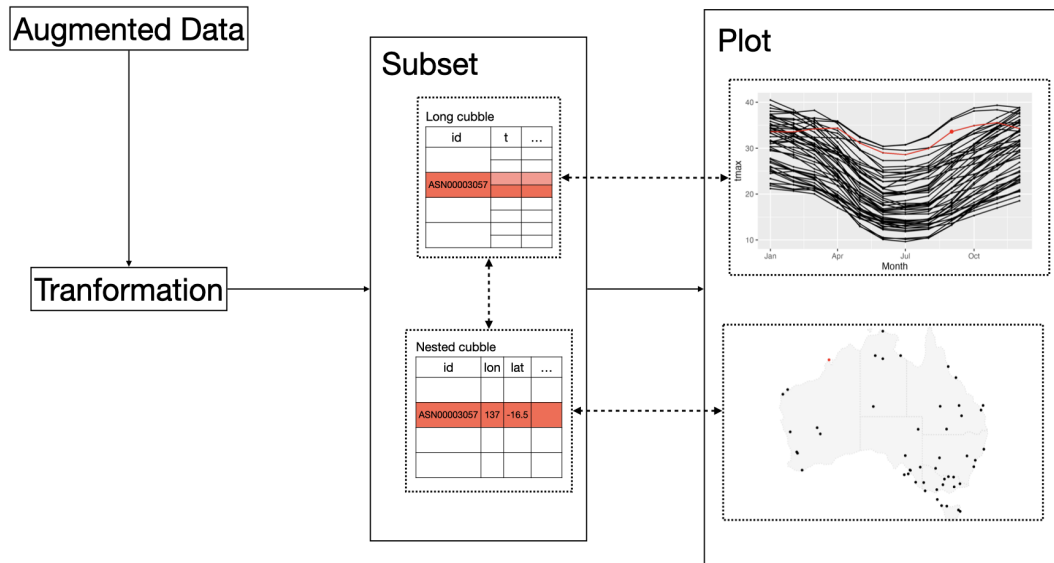


Figure 3: demon interactivity

## 4. Examples

### 4.1. Australia precipitation pattern in 2020

#### *Glyph map*

This vignette introduces how to perform spatial and temporal manipulation in a cubble with `dplyr` verbs. We will illustrate with `weatherdata::historical_tmax` data, which have the historical maximum temperature recorded for Australian stations with the earliest dating back to 1859.

`historical_tmax` is already in a nested cubble format, which is suitable for station-wise manipulation. `rnoaa` construct the station id by prefix `ASN00` to the Bureau of Meteorology (BOM) station number. In **BOM's numbering system**, the 2nd and 3rd digit denotes the state a station is located in and this is equivalent to the 7th to 8th digit in our string. We can mutate/ filter station in a particular state based on this information and here we add a column called `state_id` and filter out the ones in New South Wales and Victoria (ranging from 46 to 90).

```
R> tmax_nested <- weatherdata::historical_tmax %>%
+   mutate(state_id = stringr::str_sub(id, 7, 8)) %>%
+   filter(between(state_id, 46, 90))
```

There are some operations in the time dimension we would like to make:



- Extract observations in a particular period, say, those in 1971 to 1975 and 2016 to 2020. This can be used to compare the historical and recent climate.
- Summarise daily records into monthly to remove sparsity

These time dimension operations can be computed in the long form and `stretch()` converts a nested cubble to a long cubble. From a long cubble, we can write the exact same dplyr codes to complete the two tasks:

```
R> tmax_long <- tmax_nested %>%
+   stretch() %>%
+   filter(lubridate::year(date) %in% c(1971:1975, 2016:2020)) %>%
+   mutate(
+     month = lubridate::month(date),
+     group = as.factor(ifelse(
+       lubridate::year(date) > 2015,
+       "2016 ~ 2020",
+       "1971 ~ 1975"
+     ))
+   ) %>%
+   group_by(month, group) %>%
+   summarise(tmax = mean(tmax, na.rm = TRUE))
```

A data quality issue with the `rnoaa` data is that while it records the first and last year recorded of each series, it doesn't report the period of missingness. For example, station `ASN00047048` starts its first record in 1957, pauses for a period from 1963 to 1990, and then resumes its recording till today. Out of the 77 stations in New South Wales and Victoria, 7 stations have this issue and we would like to remove those stations from the comparison.

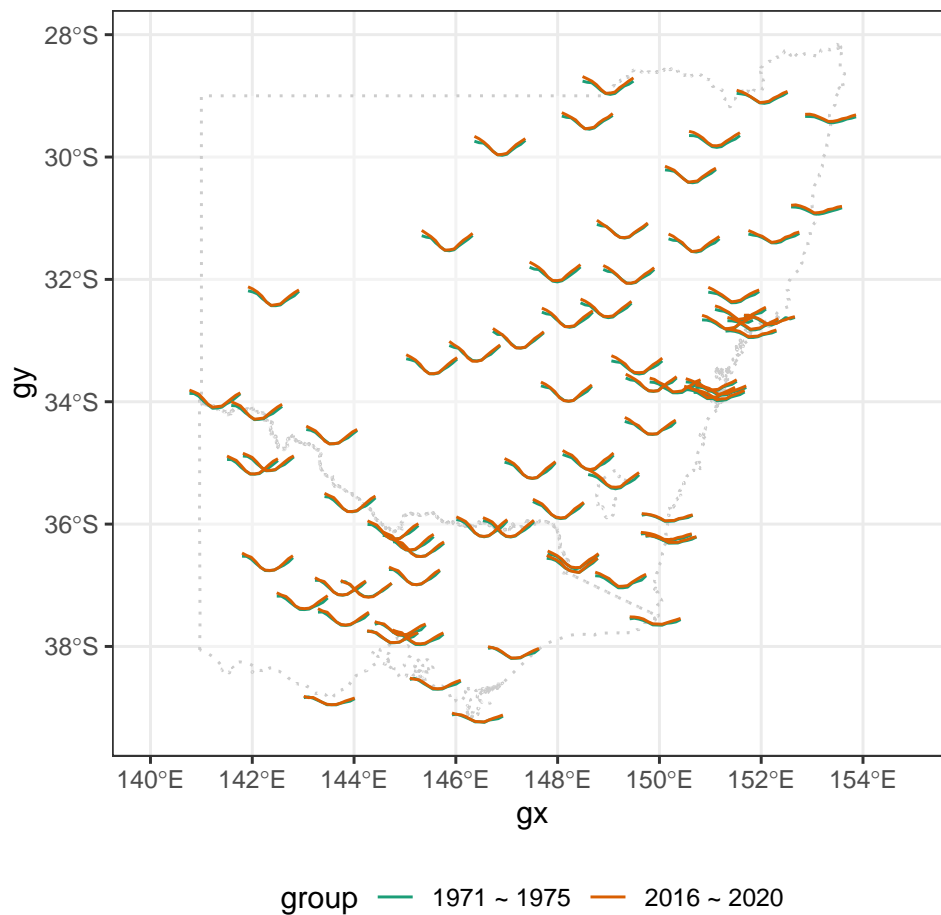
Again, this is a station-wise operation and to convert back from the long cubble to a nested one, use `tamp()`. Here we keep the stations with 24 observations (12 months for both periods) after the monthly aggregation.

```
R> tmax_nested2 <- tmax_long %>%
+   tamp() %>%
+   filter(nrow(ts) == 24)
```

In some visualisation, we may need information from both spatial and temporal dimension. One example of this is a glyph map, where spatial variables, i.e. `longitude` and `latitude`, are used to construct the major axes and temporal variables, i.e. `month` and `tmax`, are used to construct the minor axes. This requires these variables to be in the same table, rather than in different forms. In cubble, you can append spatial variables that are invariant to the `key` with `migrate()`

```
R> tmax_long2 <- tmax_nested2 %>%
+   stretch() %>%
+   migrate(latitude, longitude)
```

Now it's time for the glyph map!



### Aggregation

On top of `aus_climate` data in this package, `climate_full` from package `weatherdata` has daily climate data of 639 Australia stations from 2016 to 2020. This is where these stations locate in an Australia map:

This is a lot of stations to look at for one climate variable and they can't all fit into a glyph map. What we can do is to group stations into clusters and look at the aggregated series in the glyph map. In this vignette, I will introduce how to perform spatial (and temporal) aggregation using hierarchical data structure in `cubble`.

First let's add a new column `ll` for calculating distance matrix in the next section and aggregate the daily precipitation into weekly measure. These steps should be familiar from the vignette *Data manipulation with cubble*.

```
R> station_nested <- weatherdata::climate_full %>%
+   mutate(ll = s2::s2_lnglat(long, lat)) %>%
+   stretch() %>%
+   mutate(wk = lubridate::week(date)) %>%
+   group_by(wk) %>%
+   summarise(prcp = sum(prcp, na.rm = TRUE)) %>%
+   tamp()
```

Imposing a clustering structure can be thought of as building a hierarchical structure where stations are nested within clusters. As an example to illustrate here, we use a kmean clustering algorithm based on the distance matrix and specify the number of centres to be 20. More complex algorithms can also be used for more complex problem, as long as a mapping from each station id to the cluster id can be constructed. We then join this data to our station data:

```
R> dist_raw <- scale(s2::s2_distance_matrix(station_nested$ll, station_nested$ll))
R>
R> cluster_res <- tibble(
+   id = station_nested$id,
+   cluster = kmeans(dist_raw, centers = 20, nstart = 50)$cluster
+ )
R>
R> station_nested <- station_nested %>%
+   left_join(cluster_res)
```

One thing we hope to do with the cluster is to find the coordinates of the centroid. These are variables variant to the station but invariant to the cluster and it would be nice to have a function that structure each cluster as a row. `switch_key()` is the function that does this: it lets you to specify a new key, say `cluster` and nests all spatial variables variant to `cluster` into a column. Temporal observations from different stations while within the same cluster are bound in the nested column `ts`.

```
R> cluster_nested <- station_nested %>%
+   switch_key(cluster)
```

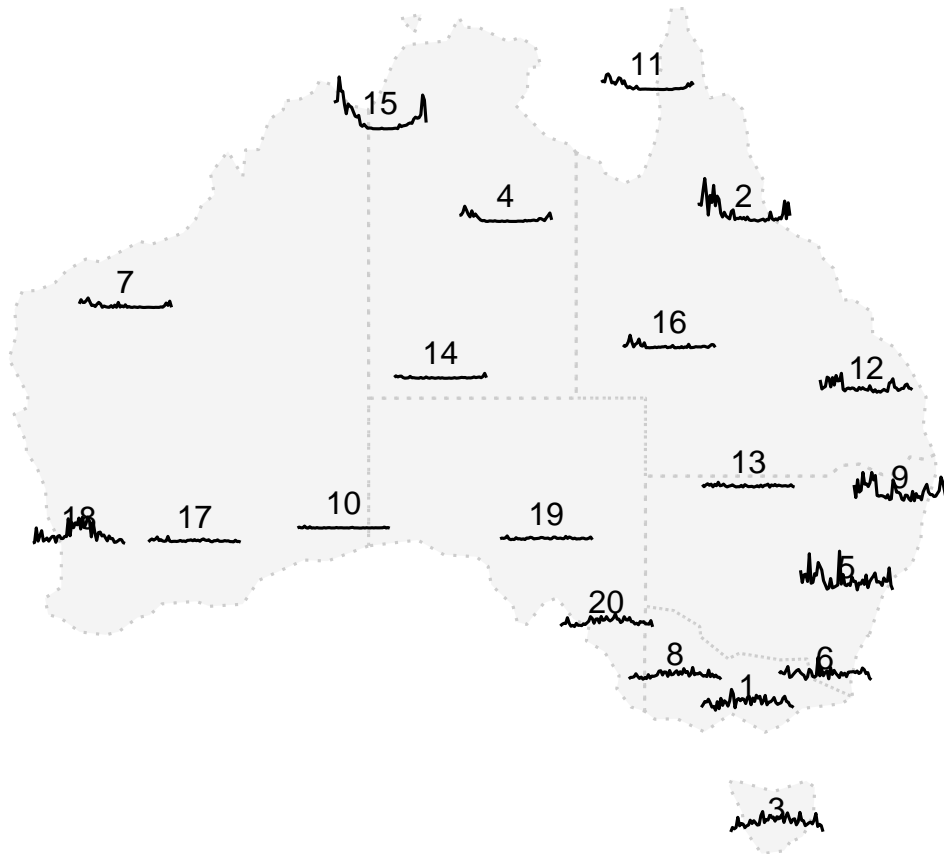
This structure makes it easy to compute cluster level variable, although there are a few steps to calculate the centroid coordinates: we need to find the convex hull that wraps around the cluster, make it a polygon, find the centroid of the polygon and finally, extract the x and y coordinate of each centroid:

```
R> cluster_nested <- cluster_nested %>%
+   mutate(
+     chull = list(chull(.val$long, .val$lat)),
+     ll_cluster = sf::st_as_sfc(
+       s2::s2_make_polygon(c(.val$long[chull]),
+         c(.val$lat[chull]),
+         oriented = FALSE
+       )
+     ),
+     centroid = s2::s2_centroid(ll_cluster),
+     cent_long = s2::s2_x(centroid),
+     cent_lat = s2::s2_y(centroid)
+   )
```

After we have got `cluster_nested`, spatial and temporal data at both levels can be easily obtained. If we use `station` and `cluster` prefix to denote the two levels and `nested` and `long` for whether the data shows the spatial or temporal dimension, the relationship among the four datasets can be illustrated in the following workflow:

Start with the original `station_nested`, `stretch()` expands the `ts` column with each station (`id`) forming a group and attach variables invariant to `id` as an attribute. `switch_key()` changes the `key` from `id` to `cluster` and nests all the spatial variables that variant to `cluster`. `stretch()` `cluster_nested` will store variables that are invariant to `cluster` as a tibble in the attribute.

Now we can obtain the aggregated series as described in the workflow diagram above and construct the glyph map with `GGally::glyphs()`:



We can also look at the precipitation of each individual station within the same cluster:

Lastly, there is one series on Tasmania island standing out from others, lets look at where it is:

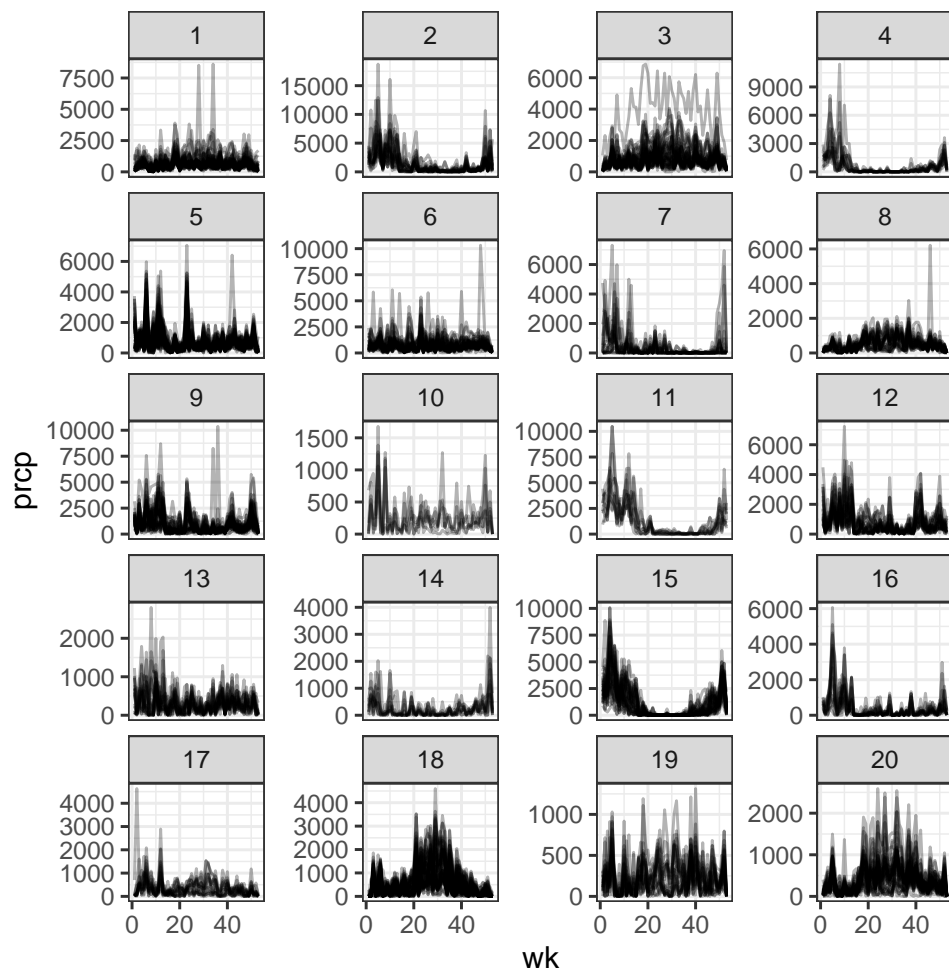


Figure 4: shtishisa



## 4.2. River level data in Victoria water gauges

The water level data comes from [Bureau of Meteorology](#) and has a copy in `weatherdata`. Here we extract the water course level and add a column annotate this data of type `river`. For the rainfall data, we will still use the `weatherdata::climate_full`, filtering for Victorian stations in 2020 should be pretty familiar by now. Again, we first look at where these stations are on the map first:

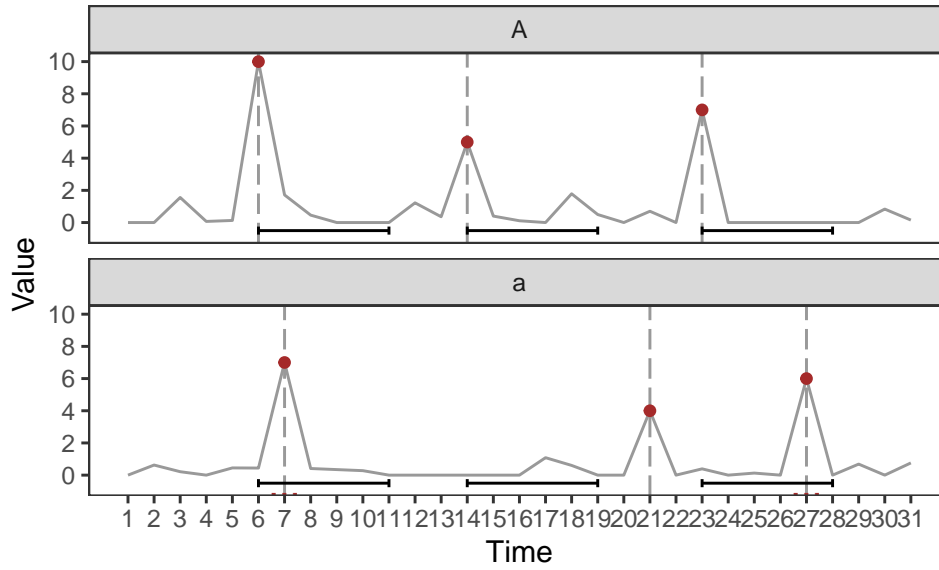
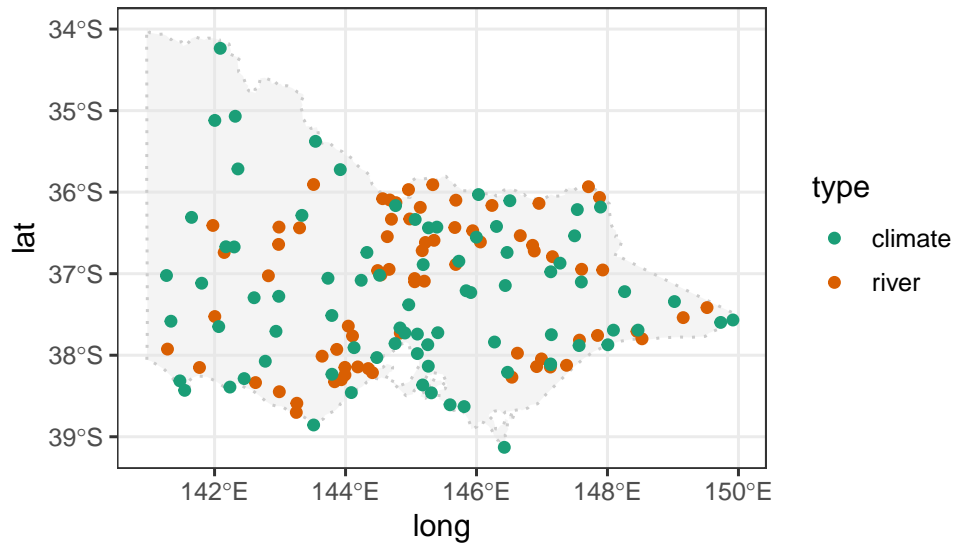


Figure 5: shtishisaasdf



Temporal matching checks how spatially matched pairs align temporally. We use the following chart to illustrate how the temporal matching works:

For each spatially matched pair, say *A* and *a*, we first find the largest *n* points in each series, colored in brown points here. Here we use the largest three but you can tune this number by `temporal_n_highest`. Then we construct the interval of the largest points from one series and see how many points, from the other series, fall into the intervals. The series used to construct the interval is controlled by `temporal_independent` and the window size by `temporal_window` with a default of 5.

In this illustration, we construct the interval based on series *A* and two of the three peaks from *a* falls into this interval at Time 7 and 27.

There's another mandatory argument that hasn't been introduced above: `temporal_var_to_match`.

This argument controls the variable to match and it needs to appear in both the `major` and `minor` set. In the water level matching example, we match the variable `Water_course_level` from `river` to `prcp` from `climate`, hence need to manually rename one of them to match the other, here we rename `Water_course_level` to `prcp` in `river`:

```
R> river <- river %>%
+   stretch() %>%
+   rename(prcp = Water_course_level) %>%
+   tamp()
```

Now we use `match_sites()` to first pair the weather stations with the river gauges spatially and then apply the temporal matching on `prcp`. We will construct the interval based on peaks in `climate` since we would expect a lag effect for precipitation to flow into the river and cause a raise in river level, hence `temporal_independent = climate`. We select the 30 highest peak from the series to construct the match by setting `temporal_n_highest = 30`. This is a tuning parameter and you can start with 10% of the points of one series (here we have daily data for a year, 10% is roughly 30 points). `temporal_min_match` filters out pairs don't have enough match and to return all the pairs, set `temporal_min_match` to 0.

```
R> res <- match_sites(river, climate,
+   temporal_var_to_match = prcp,
+   temporal_independent = climate,
+   temporal_n_highest = 30,
+   temporal_min_match = 15
+ )
R>
R> res
```

```
# cubbl:   id [8]: nested form
# bbox:    [144.52, -37.73, 146.06, -36.55]
# temporal: date [date], prcp [dbl]
  id      name      lat long type  ts      .dist .group n_match
  <chr>   <chr>      <dbl> <dbl> <chr> <list>  <dbl>  <int>  <int>
1 405234   SEVEN CREEKS @ D~ -36.9  146. river <tibble ~ 6.15    5    21
2 ASN00082042 strathbogie   -36.8  146. clim~ <tibble ~ 6.15    5    21
3 404207   HOLLAND CREEK @ ~ -36.6  146. river <tibble ~ 8.54   10    21
4 ASN00082170 benalla airport -36.6  146. clim~ <tibble ~ 8.54   10    21
5 230200   MARIBYRNONG RIVE~ -37.7  145. river <tibble ~ 6.17    6    19
6 ASN00086038 essendon airport -37.7  145. clim~ <tibble ~ 6.17    6    19
7 406213   CAMPASPE RIVER @~ -37.0  145. river <tibble ~ 1.84    1    18
8 ASN00088051 redesdale   -37.0  145. clim~ <tibble ~ 1.84    1    18
```

The output from temporal matching is also a cubbl, with additional column `.dist` and `.group` inherent from spatial matching and `n_match` for the number of matched temporal peaks. Then you can use this output to plot the location of match or to look at the series:

### 4.3. ERA5: climate reanalysis data



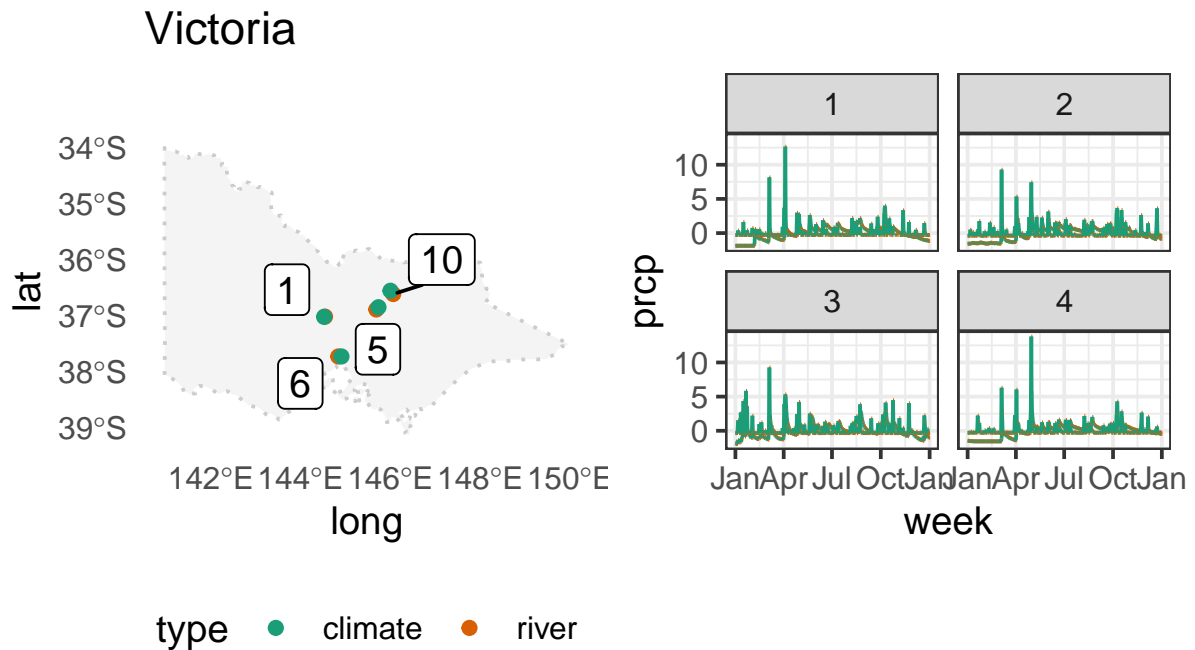
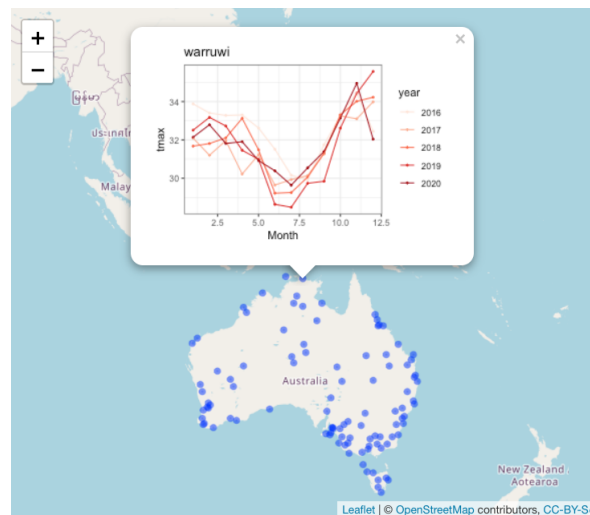
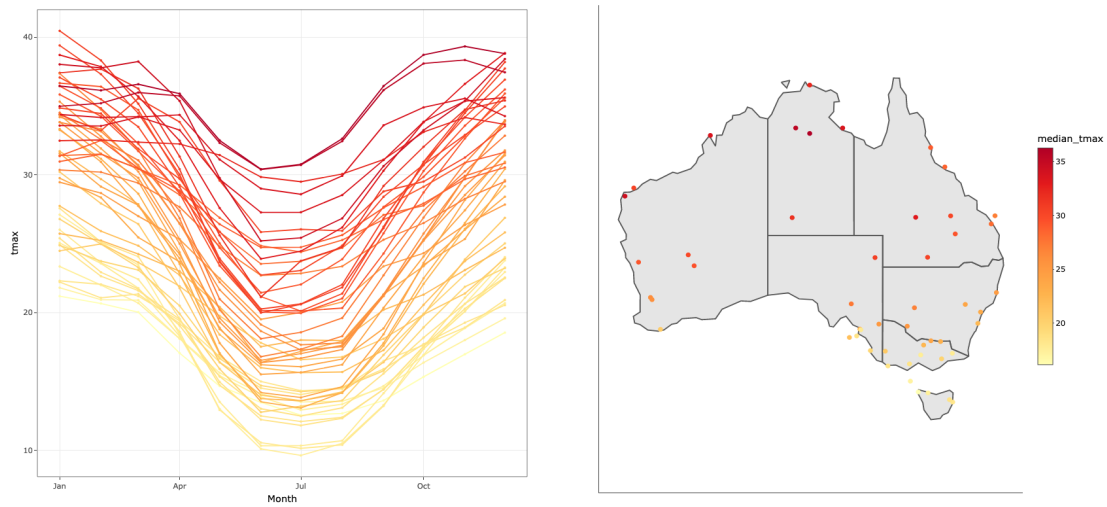


Figure 6: ...

```
R> nc <- ncdf4::nc_open(here::here("data/adaptor.mars.internal-1638344933.932894-4194-11-a
R>
R> nc <- ncdf4::nc_open(here::here("data/tas_Amon_ACCESS-CM2_ssp585_r1i1p1f1_gn_20150116-2
R> temp <- as_cubble(nc, vars = "tas")
R> vic_map <- ozmaps::abs_ste %>% filter(NAME == "Victoria")
R>
R> res <- temp %>%
+   stretch(ts) %>%
+   mutate(d2m = d2m - 273.15) %>%
+   filter(lubridate::hour(time) == 12) %>%
+   group_by(id) %>%
+   summarise(d2m = mean(d2m, na.rm= TRUE)) %>%
+   migrate(long, lat)
R>
R> vic_string <- vic_map %>% sf::st_cast("MULTILINESTRING")
R>
R> res %>%
+   ggplot() +
+   geom_tile(aes(x = long, y = lat, fill = d2m)) +
+   geom_sf(data = vic_string, alpha = 0.5, fill = "transparent") +
+   theme_bw() +
+   colorspace::scale_fill_continuous_sequential("Mint")
```

#### 4.4. Interactive graphic with cubble



## 5. Conclusion

## 6. Old stuff

In the temporal aspect, the `tsibble` (?) structure and its tidyverts ecosystem have provided a [...] workflow to work with temporal data. In a `tsibble` structure, temporal data is characterised by `index` and `key` where `index` is the temporal identifier and `key` is the identifier for multiple series, which could be used as a spatio identifier. However, a `tsibble` object, by construction, always requires the `index` in its structure. This makes it less appealing for spatio-temporal data since the output of calculated spatio-specific variables (i.e. features of each series) don't have the time dimension. Analysts will either need to have an additional step to join this output to the original `tsibble` or operate with variables stored in two separate objects. In addition, the long form structure of a `tsibble` object means spatio variables (i.e. longitude, latitude, and features of each series if joined back to the `tsibble`) of each spatio identifier will be repetitively recorded at each timestamp. This repetition is unnecessary and would inflate the object size for long series.

## 7. Create a cubble

The creation of a `cubble` requires the site identifier (`key`), as well as the spatial (`coords`) and temporal (`index`) identifier. `climate_flat` is already a `tibble` and it uses `id` to identify each station, `date` as the time identifier, and `c(long, lat)` as the spatial identifier. To create a `cubble` for this data, use:

```
R> climate_flat %>% as_cubble(key = id, index = date, coords = c(long, lat))
```

```
# cubble:   id [5]: nested form
# bbox:     [115.97, -32.94, 133.55, -12.42]- check gap on long and lat
# temporal: date [date], prcp [dbl], tmax [dbl], tmin [dbl]
  id      lat long elev name      wmo_id ts
<chr>    <dbl> <dbl> <dbl> <chr>    <dbl> <list>
1 ASN00009021 -31.9 116. 15.4 perth airport 94610 <tibble [366 x 4]>
2 ASN00010311 -31.9 117. 179  york          94623 <tibble [366 x 4]>
3 ASN00010614 -32.9 117. 338  narrogin     94627 <tibble [366 x 4]>
4 ASN00014015 -12.4 131. 30.4 darwin airport 94120 <tibble [366 x 4]>
5 ASN00015131 -17.6 134. 220  elliot      94236 <tibble [366 x 4]>
```

Most of the time, spatio-temporal data doesn't come into this form and analysts need to query the climate variables based on station metadata. **This is also a problem illustrated in Section 3.5 in @tidydata. Here we provide a structured way to query this data based on the row-wise operator and nested list.** For this type of task, one can structure a metadata into a `tibble` and use row-wise operator to query the climate variables into a nested list. As an example here we demonstrate the workflow to find the 5 closest stations to Melbourne. We first create a station data frame with the 5 target stations.

```
# A tibble: 5 x 8
  id      lat long elev name      wmo_id dist city
<chr>    <dbl> <dbl> <dbl> <chr>    <dbl> <dbl> <chr>
```

```

1 ASN00086038 -37.7 145. 78.4 essendon airport      95866 10.8 melbourne
2 ASN00086282 -37.7 145. 113. melbourne airport    94866 20.1 melbourne
3 ASN00086077 -38.0 145. 12.1 moorabbin airport     94870 21.9 melbourne
4 ASN00088162 -37.4 145. 528. wallan (kilmore gap)  94860 48.1 melbourne
5 ASN00087113 -38.0 144. 10.6 avalon airport       94854 48.8 melbourne

```

We can query the climate information into a nested list named `ts` for each station with the `rowwise()` operator. To create a cubble, supply the same identifiers as with the first example.

```

R> sydmel_climate <- stations %>%
+   rowwise() %>%
+   mutate(ts = list(meteo_pull_monitors(id,
+     date_min = "2020-01-01",
+     date_max = "2020-12-31",
+     var = c("PRCP", "TMAX", "TMIN")
+   ) %>%
+     select(-id))) %>%
+   as_cubbl(key = id, index = date, coords = c(long, lat))

# cubbl:   id [5]: nested form
# bbox:    [144.47, -38.03, 145.1, -37.38]
# temporal: date [date], prcp [dbl], tmax [dbl], tmin [dbl]
  id          lat long elev name          wmo_id dist city  ts
  <chr>        <dbl> <dbl> <dbl> <chr>        <dbl> <dbl> <chr>  <list>
1 ASN00086038 -37.7 145. 78.4 essendon airport    95866 10.8 melbo~ <tibbl~
2 ASN00086282 -37.7 145. 113. melbourne airport    94866 20.1 melbo~ <tibbl~
3 ASN00086077 -38.0 145. 12.1 moorabbin airport     94870 21.9 melbo~ <tibbl~
4 ASN00088162 -37.4 145. 528. wallan (kilmore gap)  94860 48.1 melbo~ <tibbl~
5 ASN00087113 -38.0 144. 10.6 avalon airport       94854 48.8 melbo~ <tibbl~

```

Below are the how the nested and long form look like for Australia climate data, which records daily precipitation, maximum and minimum temperature for 55 stations across Australia from 2015- 2020. Notice that each station forms a group in both forms and specifically, the nested and long form have a underlying `rowwise_df` and `grouped_df` respectively.

With a cubic framework on mind, different types of manipulation with cubbl can be thought of as slicing the cube in various way. The table below shows how some `dplyr` verbs are mapped into the operation in a cubbl. With the existing grouping on the station, additional grouping can be added with `group_by` and removed with `ungrouped`. [talk about why it is useful]

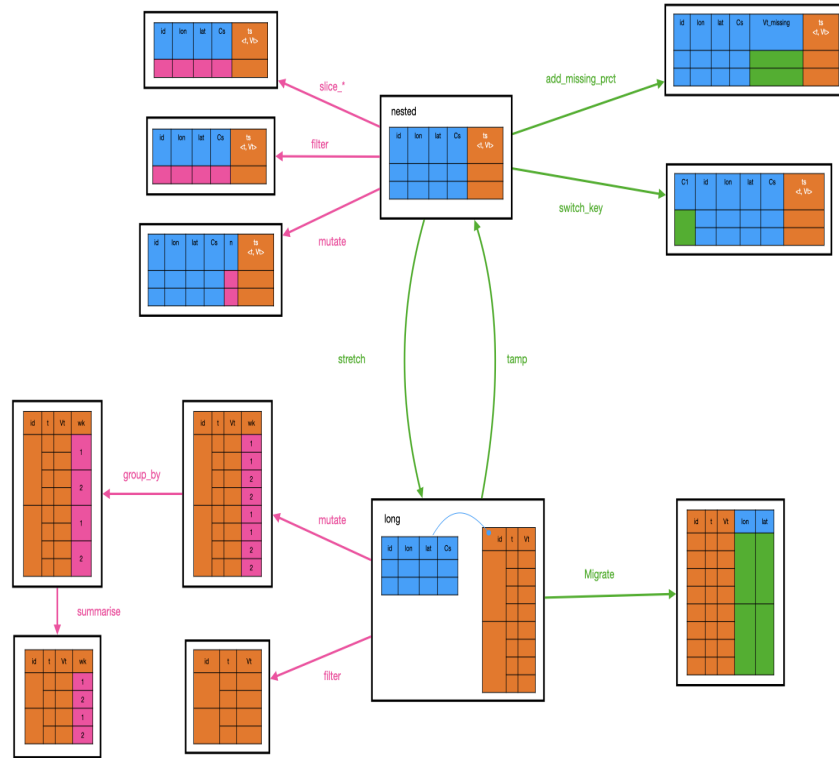


Figure 7: Cubble operations

## 7.1. Cubble operations

## References

- Baddeley A, Turner R (2005). “Spatstat: An R Package for Analyzing Spatial Point Patterns.” *Journal of Statistical Software*, **12**(6), 1–42. URL <https://doi.org/10.18637/jss.v012.i06>.
- Buja A, Asimov D, Hurley C (1988). “Elements of a viewing pipeline.” *Dynamic Graphics Statistics*, p. 277.
- Buja A, Cook D, Swayne DF (1996). “Interactive high-dimensional data visualization.” *Journal of computational and graphical statistics*, **5**(1), 78–99. URL <https://doi.org/10.2307/1390754>.
- Cheng J, Karambelkar B, Xie Y (2021). *leaflet: Create Interactive Web Maps with the JavaScript 'Leaflet' Library*. R package version 2.0.4.1, URL <https://CRAN.R-project.org/package=leaflet>.

- Cheng J, Sievert C (2021). *crosstalk: Inter-Widget Interactivity for HTML Widgets*. R package version 1.1.1, URL <https://CRAN.R-project.org/package=crosstalk>.
- Cheng X, Cook D, Hofmann H (2016). “Enabling interactivity on displays of multivariate time series and longitudinal data.” *Journal of Computational and Graphical Statistics*, **25**(4), 1057–1076. URL <https://doi.org/10.1080/10618600.2015.1105749>.
- Gohel D, Skintzos P (2021). *ggiraph: Make 'ggplot2' Graphics Interactive*. R package version 0.7.10, URL <https://CRAN.R-project.org/package=ggiraph>.
- Pebesma E (2012). “spacetime: Spatio-Temporal Data in R.” *Journal of Statistical Software*, **51**(7), 1–30. URL <https://doi.org/10.18637/jss.v051.i07>.
- Pebesma E (2021). *stars: Spatiotemporal Arrays, Raster and Vector Data Cubes*. R package version 0.5-2, URL <https://CRAN.R-project.org/package=stars>.
- Pebesma E, Bivand RS (2005). “S classes and methods for spatial data: the sp package.” *R news*, **5**(2), 9–13.
- Pebesma EJ (2018). “Simple features for R: standardized support for spatial vector data.” *R Journal*, **10**(1), 439.
- Ryan JA, Ulrich JM (2020). *xts: eXtensible Time Series*. R package version 0.12.1, URL <https://CRAN.R-project.org/package=xts>.
- Sievert C (2020). *Interactive Web-Based Data Visualization with R, plotly, and shiny*. Chapman and Hall/CRC. ISBN 9781138331457. URL <https://plotly-r.com>.
- Sutherland P, Rossini A, Lumley T, Lewin-Koh N, Dickerson J, Cox Z, Cook D (2000). “Orca: A visualization toolkit for high-dimensional data.” *Journal of Computational and Graphical Statistics*, **9**(3), 509–529. URL <https://www.tandfonline.com/doi/abs/10.1080/10618600.2000.10474896>.
- Wang E, Cook D (2020). *tsibbletalk: Interactive Graphics for Tsibble Objects*. R package version 0.1.0, URL <https://CRAN.R-project.org/package=tsibbletalk>.
- Wang E, Cook D, Hyndman RJ (2020). “A new tidy data structure to support exploration and modeling of temporal data.” *Journal of Computational and Graphical Statistics*, **29**(3), 466–478. URL <https://doi.org/10.1080/10618600.2019.1695624>.
- Wickham H (2014). “Tidy Data.” *Journal of Statistical Software*, **59**(10), 1–23. URL <https://doi.org/10.18637/jss.v059.i10>.
- Wickham H (2020). *cubelyr: A Data Cube 'dplyr' Backend*. R package version 1.0.1, URL <https://CRAN.R-project.org/package=cubelyr>.
- Wickham H, Lawrence M, Cook D, Buja A, Hofmann H, Swayne DF (2009). “The plumbing of interactive graphics.” *Computational Statistics*, **24**(2), 207–215. URL <https://doi.org/10.1007/s00180-008-0116-x>.
- Xie Y, Hofmann H, Cheng X (2014). “Reactive programming for interactive graphics.” *Statistical Science*, pp. 201–213. URL <https://doi.org/10.1214/14-STS477>.

**Affiliation:**

---

*Journal of Statistical Software*

published by the Foundation for Open Access Statistics

MMMMMM YYYY, Volume VV, Issue II

[doi:10.18637/jss.v000.i00](https://doi.org/10.18637/jss.v000.i00)

<http://www.jstatsoft.org/>

<http://www.foastat.org/>

*Submitted:* yyyy-mm-dd

*Accepted:* yyyy-mm-dd

---