The Adventure of RedAmber - A data frame library in Ruby

Hirokazu SUZUKI

2023-05-13

RubyKaigi 2023 at Matsumoto Performing Arts Centre nagano, Japan

self.introduction

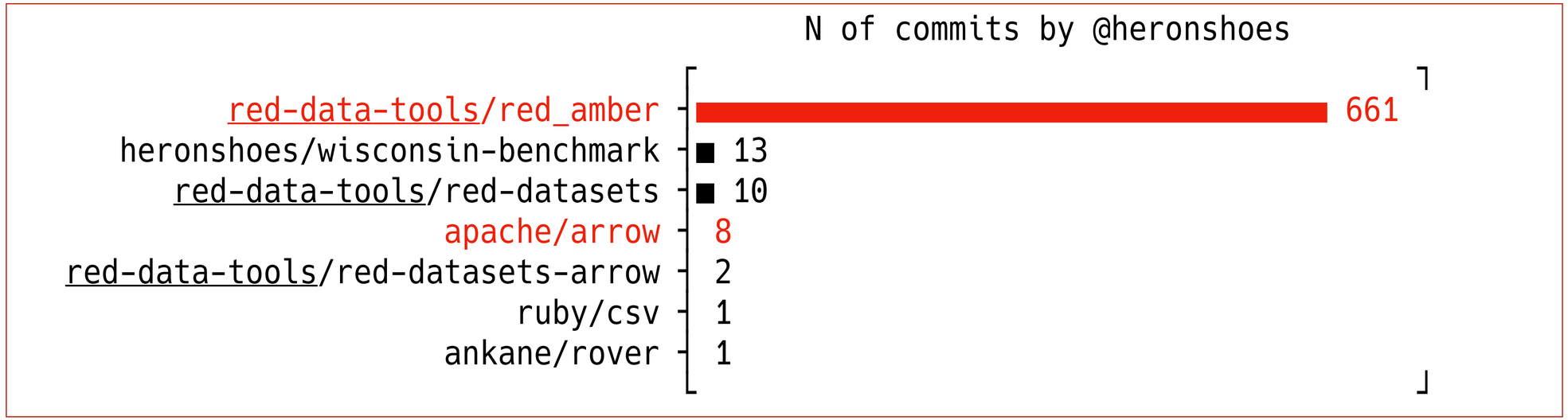
福山市
Fukuyama City

A member of
Red Data Tools

- 鈴木 弘一(Hirokazu SUZUKI)
- Github/Twitter: @heronshoes
- Living in Fukuyama city, Hiroshima, Japan
- I am an amateur Rubyist, not an IT engineer.
- I love coffee, craft beer and MINI.



My Work



Almost all the work are for RedAmber! I contribute a little to Apache Arrow.

Code for the plot above

```
require 'red_amber'
df = RedAmber::DataFrame.load(Arrow::Buffer.new(<<~CSV), format: 'csv')
  project,commit
  red-data-tools/red_amber,661
  heronshoes/wisconsin-benchmark,13
  red-data-tools/red-datasets,10
  apache/arrow,8
  red-data-tools/red-datasets-arrow,2
  ruby/csv,1
  ankane/rover,1
CSV

require ‘unicode_plot'
UnicodePlot.barplot(data: df.to_a.to_h, title: 'N of commits by @heronshoes').render
```

RedAmber ?

- RedAmber is a data frame library written in Ruby
  - Data frame is a 2D data structure
    - pandas in Python, dplyr/tidyr in R, Polars in Rust
    - Almost same as a Table in SQL
- RedAmber uses Red Arrow as its backend
  - Red Arrow is a ruby implementation in Apache Arrow project
- RedAmber was developed under the support of 2022 Ruby Association Grant

```
diamonds
  .slice { carat > 1 }
  .pick(:cut, :price)
  .group(:cut)
  .mean
  .sort('-mean(price)')
```

RedAmber::DataFrame <5 x 2 vectors>

cut	mean(price)
Ideal	8674.23
Premium	8487.25
Very Good	8340.55
Good	7753.6
Fair	7177.86

Developing RedAmber was an Adventure

- I'm a beginner in open source software development.
  - I am a Ruby user since Ruby 1.4.x
  - However, I had no git experience until February 2022 !
- It was an adventure to explore the data frame for the Rubyists.
  - Python/pandas never became a tool I was comfortable with.
- It was an adventure on the shoulder of giant, Apache Arrow !

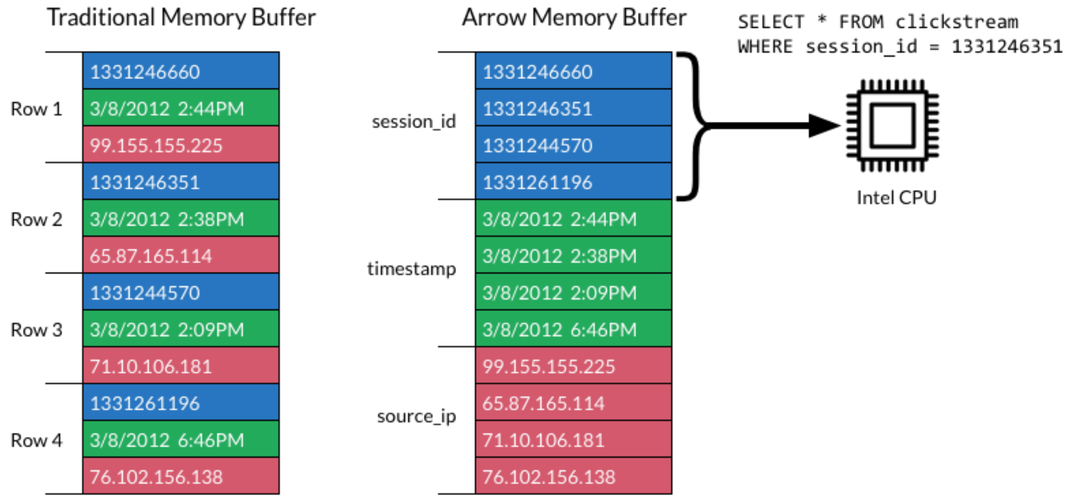
The goal of this talk is

- To introduce features of RedAmber,
  - As a library designed to provide idiomatic Ruby interface.
  - As a DataFrame for Rubyists.
- To review how I created RedAmber,
  - As a beginner in open source software development.
- To demonstrate the potential of Ruby for data processing.

Apache Arrow

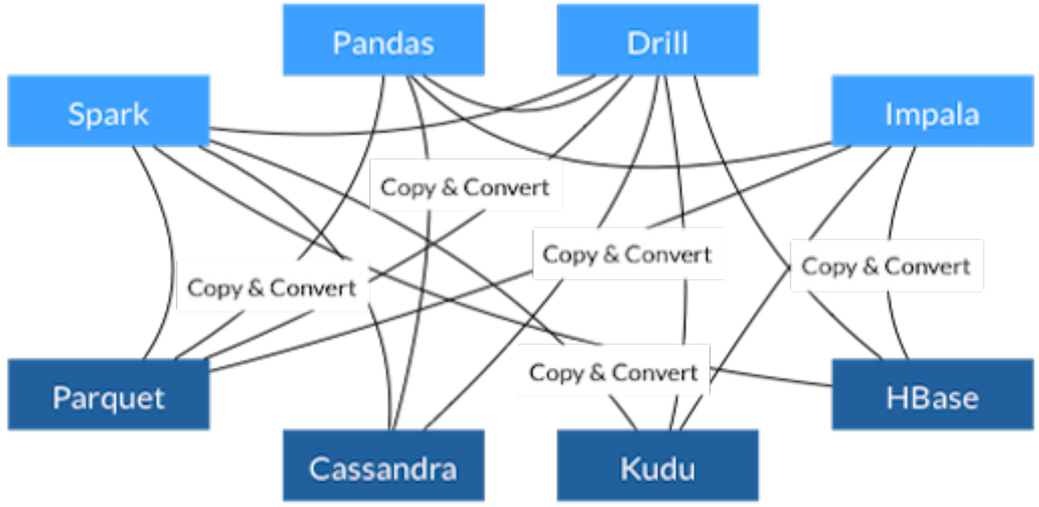
In-memory columnar format

	session_id	timestamp	source_ip
Row 1	1331246660	3/8/2012 2:44PM	99.155.155.225
Row 2	1331246351	3/8/2012 2:38PM	65.87.165.114
Row 3	1331244570	3/8/2012 2:09PM	71.10.106.181
Row 4	1331261196	3/8/2012 6:46PM	76.102.156.138



A standardized, language-agnostic specification

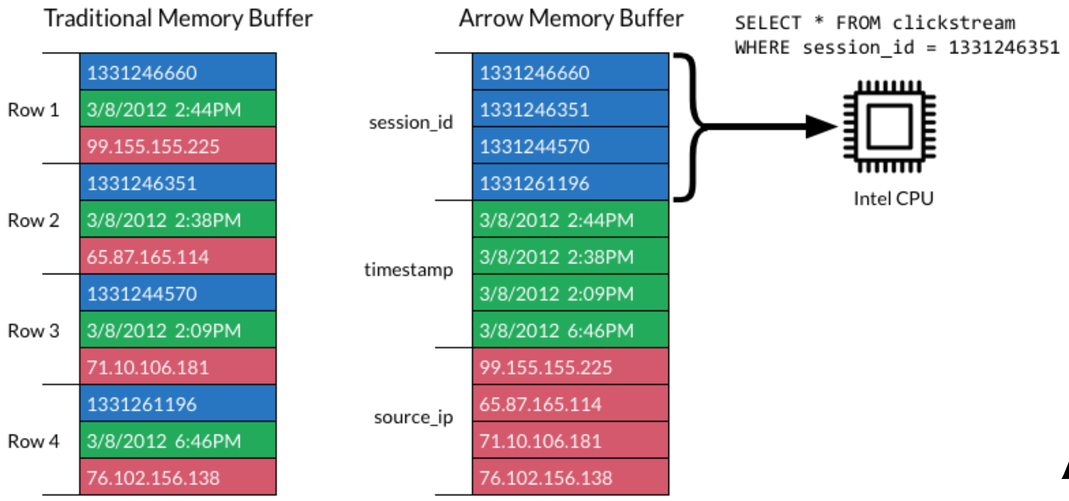
Transferring data w/o Arrow



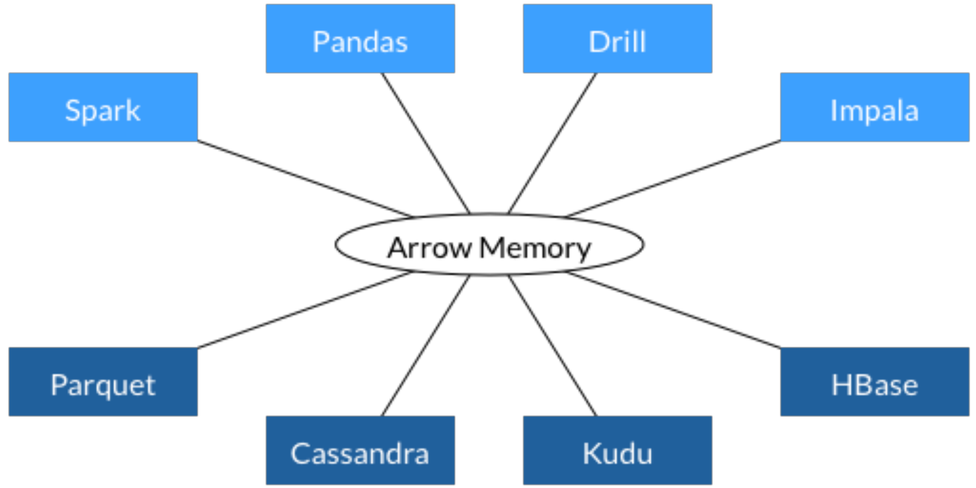
Apache Arrow

In-memory columnar format

	session_id	timestamp	source_ip
Row 1	1331246660	3/8/2012 2:44PM	99.155.155.225
Row 2	1331246351	3/8/2012 2:38PM	65.87.165.114
Row 3	1331244570	3/8/2012 2:09PM	71.10.106.181
Row 4	1331261196	3/8/2012 6:46PM	76.102.156.138



Transfer data at little-to-no cost

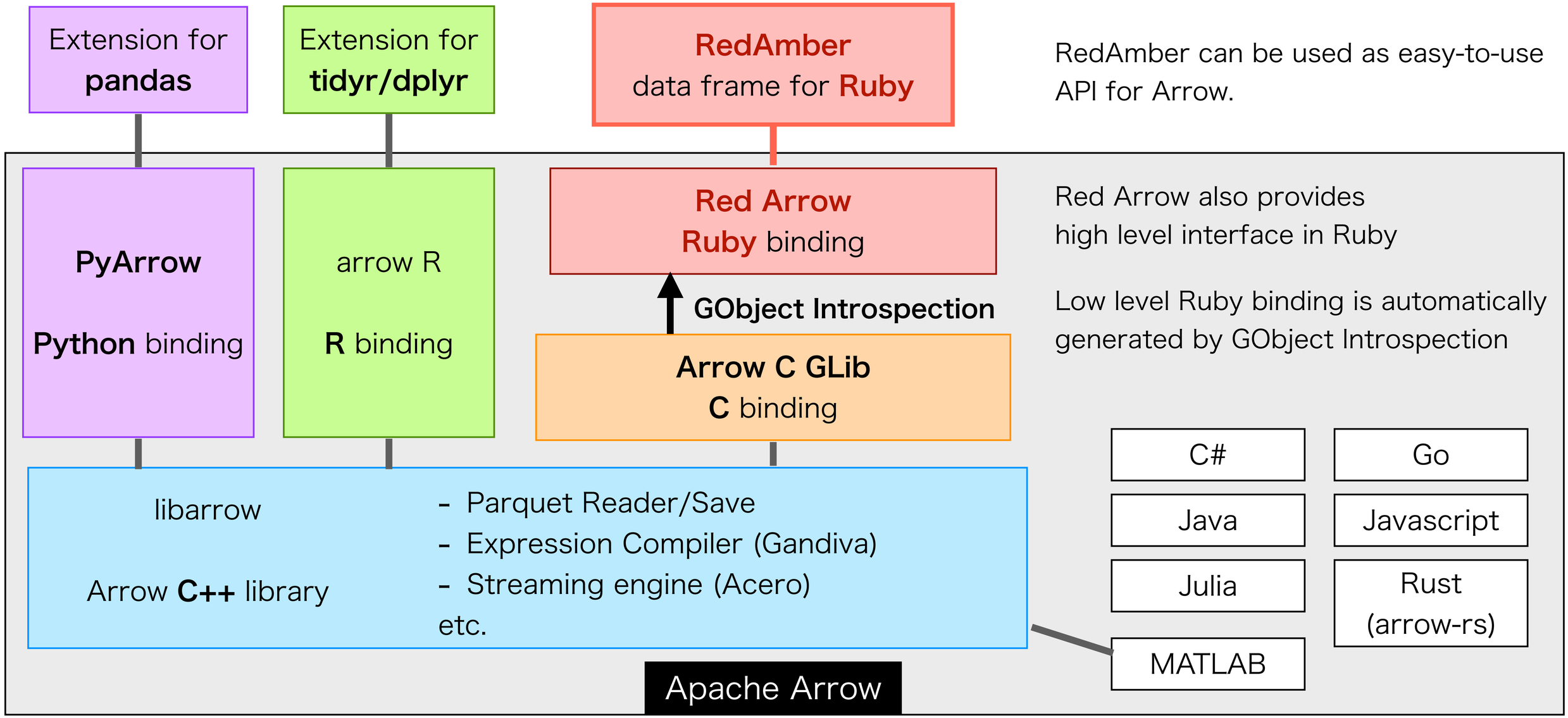


Arrow Libraries in many languages

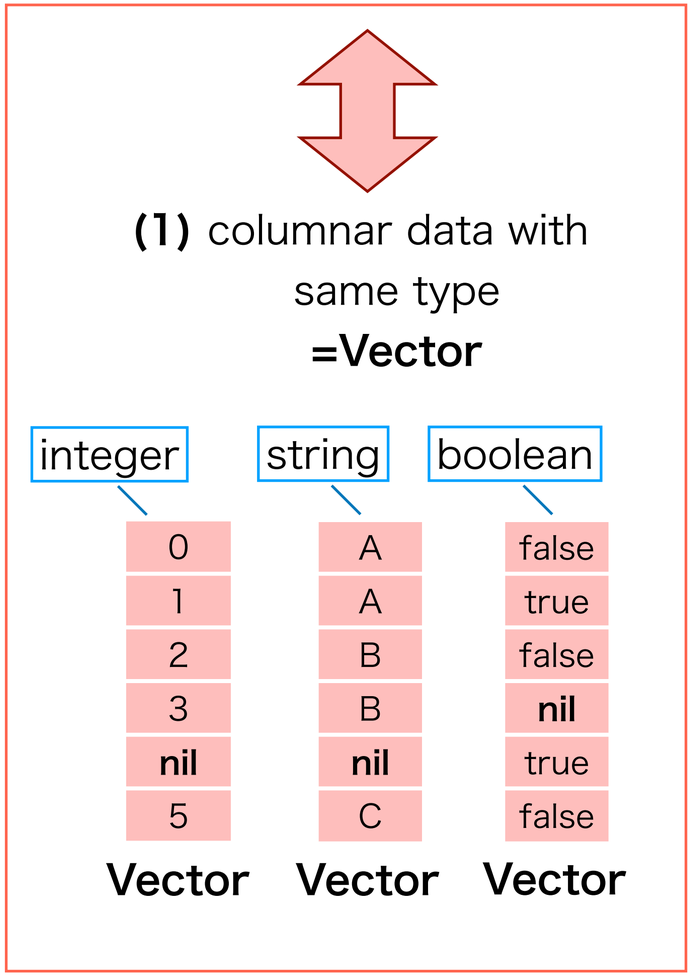
C++, C#, Go, Java, JavaScript, Julia, and Rust

C (Glib), MATLAB, Python, R, and Ruby

RedAmber on Red Arrow



DataFrame of RedAmber



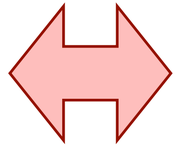
x	y	z
0	A	false
1	A	true
2	B	false
3	B	**nil**
nil	**nil**	true
5	C	false

DataFrame

(3) Column label
= **Key**
(Must be unique)

Columnar data unit :
Variable

data unit in row :
Record or
Observation



(2) Align **Vector** with same length

(4) Any column can have **nil** as a missing value

DataFrame is a data structure with;
- Efficient data handling by column
- Useful for searching and extracting records in row

Data structure in RedAmber

df

x	y	z
0	A	false
1	A	true
2	B	false
3	B	nil
nil	nil	true
5	C	false

DataFrame

```
#<RedAmber::DataFrame : 6 x 3 Vectors, 0x00000000000100a4>
         x y         z
  <uint8> <string> <boolean>
0        0 A         false
1        1 A         true
2        2 B         false
3        3 B         (nil)
4    (nil) (nil)     true
5        5 C         false
```

df.x	df.y	df.z
0	A	false
1	A	true
2	B	false
3	B	nil
nil	nil	true
5	C	false

Vector **Vector** **Vector**

```
#<RedAmber::Vector(:uint8, size=6):0x000000000000ff3c>
[0, 1, 2, 3, nil, 5]
```

```
#<RedAmber::Vector(:string, size=6):0x000000000000ff78>
["A", "A", "B", "B", nil, “C”]
```

```
#<RedAmber::Vector(:boolean, size=6):0x000000000000ff8c>
[false, true, false, nil, true, false]
```

Properties and collections of DataFrame

```
df.shape
=> [6, 3]

df.size
=> 6

df.n_keys
=> 3

df.keys
=> [:x, :y, :z]
```

df

x	y	z
0	A	false
1	A	true
2	B	false
3	B	**nil**
nil	**nil**	true
5	C	false

DataFrame

```
df.types
=> [:uint8, :string, :boolean]
```

```
df.schema
=> {:x=>:uint8, :y=>:string, :z=>:boolean}

df.vectors
=>
[#<RedAmber::Vector(:uint8, size=6):0x0000000000102e8>
[1, 2, 3, 4, nil, 6]
,
 #<RedAmber::Vector(:string, size=6):0x0000000000102fc>
["A", "A", "B", "B", nil, "C"]
,
 #<RedAmber::Vector(:boolean, size=6):0x0000000000010310>
[false, true, false, nil, true, false]
]
```

Collection methods return Ruby's Array or Hash.

=> We can use Ruby's standard way to process data.

Inside of DataFrame

df

x	y	z
0	A	false
1	A	true
2	B	false
3	B	**nil**
nil	**nil**	true
5	C	false

DataFrame

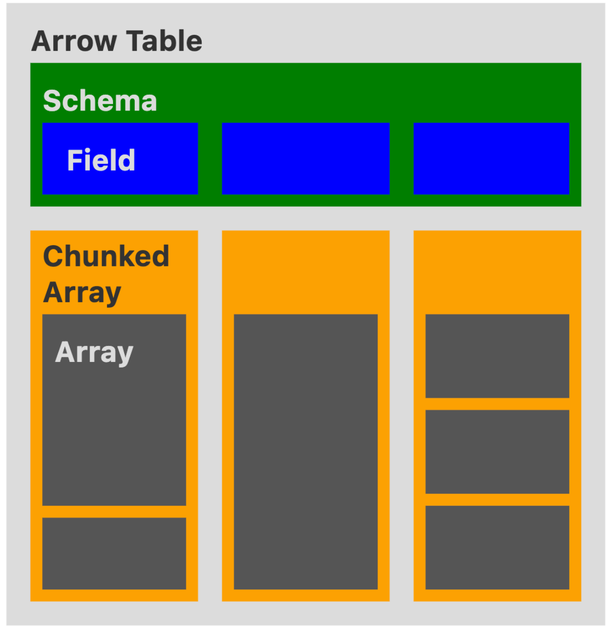
```
df.table

=>
#<Arrow::Table:0x7fe62765a418 ptr=0x7fe62d341960>
          x        y        z
0         1        A        false
1         2        A        true
2         3        B        false
3         4        B        (null)
4         5        B        true
5         6        C        false
```

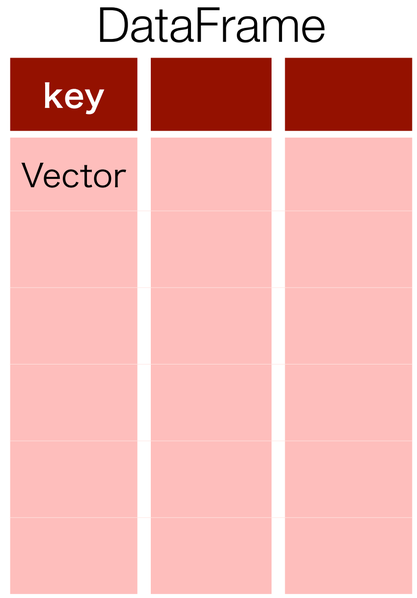
The entity of **RedAmber::DataFrame** is a Red Arrow's Table.

Difference between DataFrame and Arrow::Table

- Arrow::Table can have same column name.
  - Keys must be unique each other in RedAmber.
- Arrow::Table may have chunked Array.
  - In RedAmber, users do not need to be aware of whether the contents of the Vector are chunked.

```
#<Arrow::Table:0x1175b3840 ptr=0x7fb6da1f1ca0>
          count    name     count
0         1        A        1
1         2        B        2
2         3        C        3
```



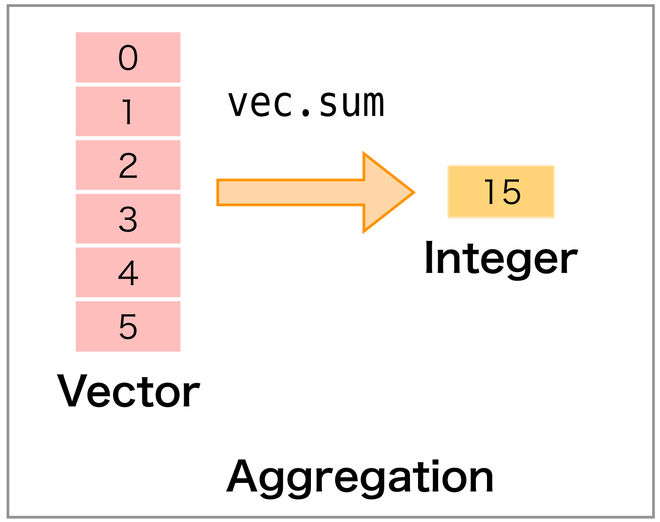


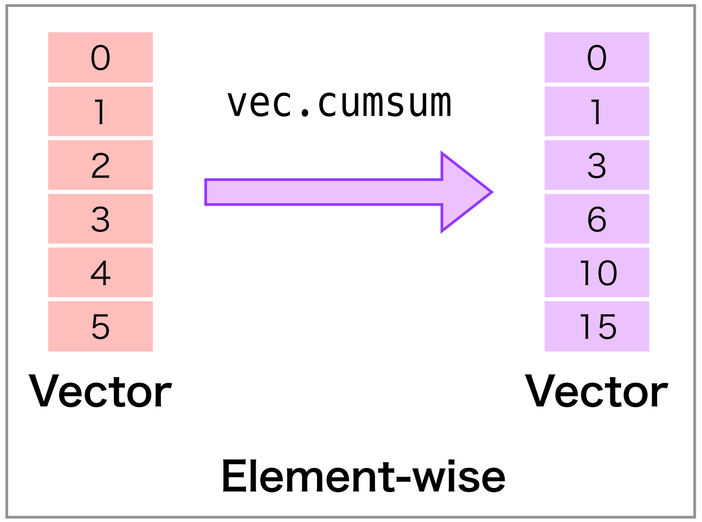
Inside of Vector

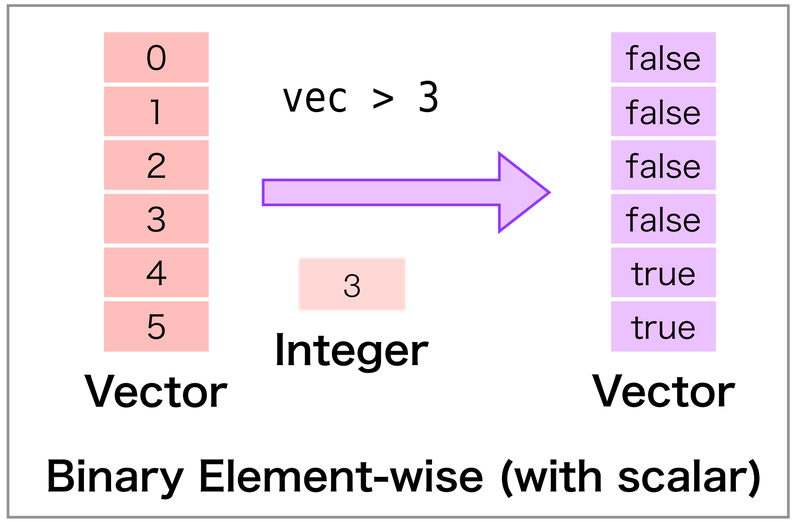

```
df.x.data

=>
#<Arrow::ChunkedArray:0x7fe629d3d300 ptr=0x7fe62743e2a0 [
  [
    1,
    2,
    3,
    4,
    5,
    6
  ]
]>
```

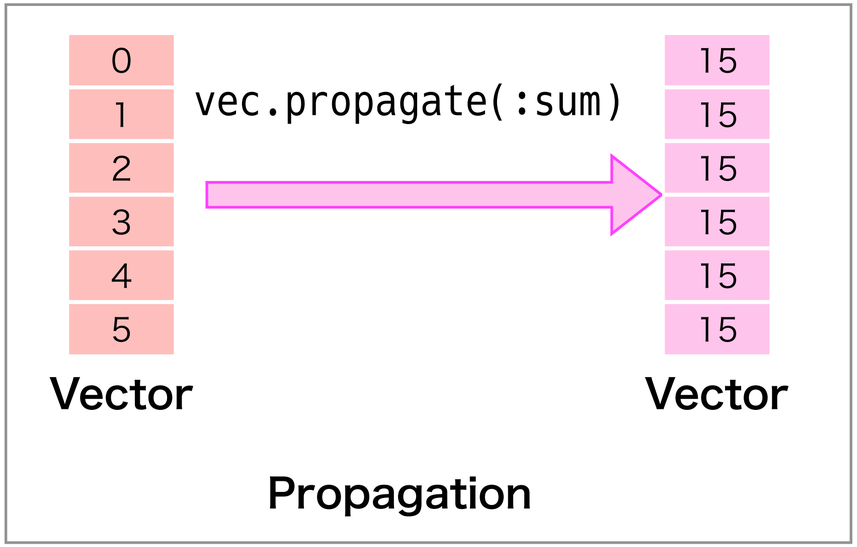
The entity of **RedAmber::Vector** is a Red Arrow's **ChunkedArray** (or a **Arrow::Array**)

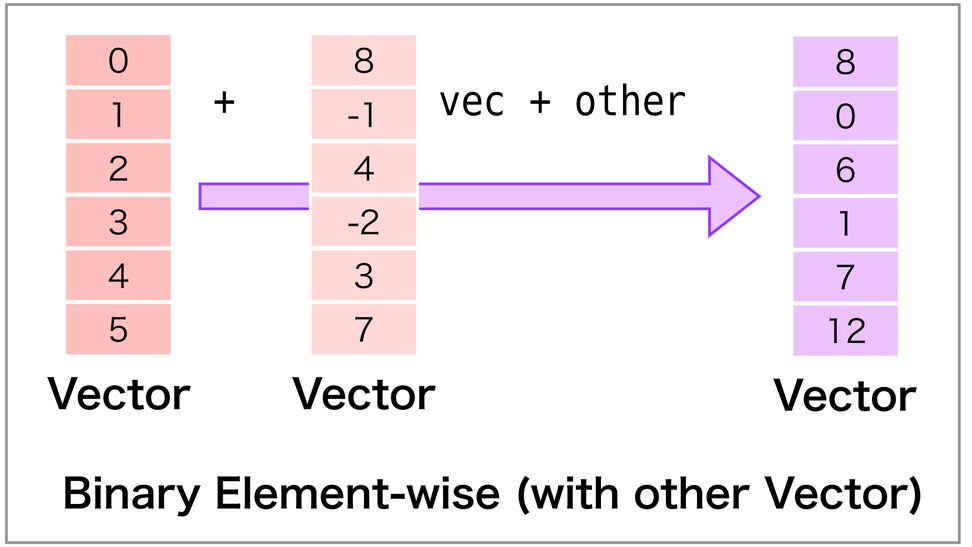
Vector's functional methods







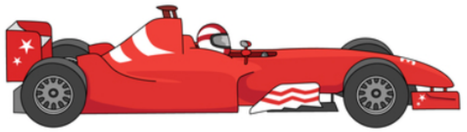




Vector has 75 methods from Arrow's C++ compute function

How I find to design DataFrame and Vector?

- I inspired by **Rover** (rover-df).
  - A data frame library in Ruby by Andrew Kane (@ankane).
  - Built on Numo::NArray.
- His development has shifted to another data frame **Polars Ruby**.
  - Blazingly fast DataFrames for Ruby



  - Powered by Polars using Apache Arrow Columnar Format as the memory model.

Creating a Vector

Create from a DataFrame

df

x	y	z
0	A	false
1	A	true
2	B	false
3	B	**nil**
nil	**nil**	true
5	C	false

DataFrame

df.x

0
1
2
3
nil
5

Vector

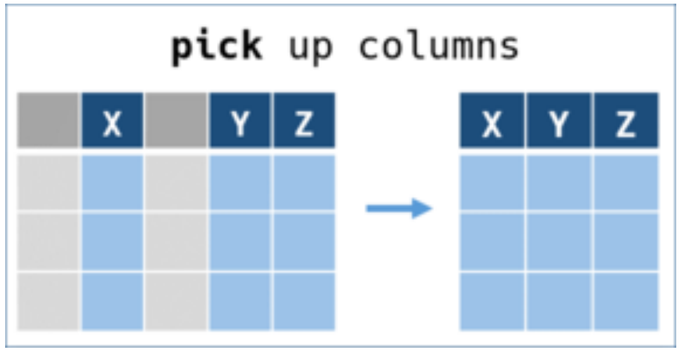
```
df.x
  - Use key name as a method via `method_missing`
  - Self can be omitted in the block
  - Unavailable :CapitalKey or :'quoted-key'

df[:x]
  - Available for all keys

df.v(:x)
  - Available for all keys
  - Self can be omitted in the block
  - A little bit faster than #[]
```

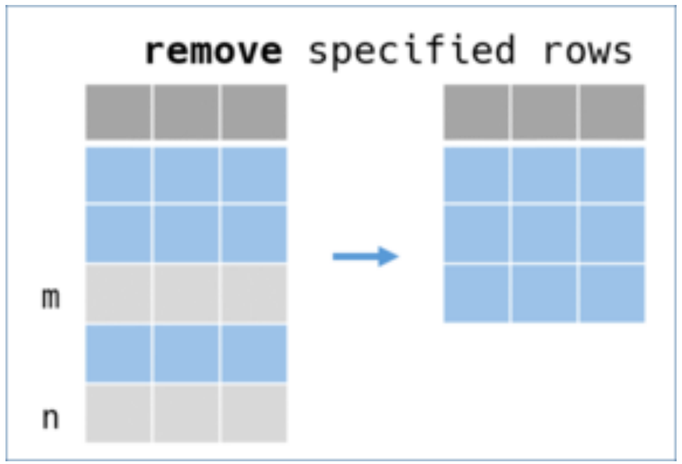
Create by a constructor

```
Vector.new(Array) or
Vector.new(Arrow::Array) or
Vector.new(Range)
```

What I did in RedAmber - (1) Orthogonal API

- Orthogonal/Complementary method pair
  - **pick** or **drop** to select/exclude columns
  - **slice** or **remove** to select/exclude rows





I separated verbs for columns and rows
because the role for columns and
for rows are different in DataFrames.

Example: Split data (1/2)

From #89 データサイエンス100本ノック(構造化データ加工編), Partially translated to alphabetical data.
Github: The-Japan-DataScientist-Society/100knocks-preprocess

We would like to **split** a customer with sales history **into training data and test data** to build a forecasting model. **Split the data randomly** in the ratio of 8:2 for each.

売上実績がある顧客を、予測モデル構築のため**学習用データとテスト用データに分割**したい。それぞれ8:2の割合で**ランダムにデータを分割**せよ。

Code

```
df_customer_selected  # view the original data
```

Output

```
#<RedAmber::DataFrame : 21971 x 10 Vectors>
          index customer_id    gender_cd gender   birth_day      age postal_cd application_store_cd application_date status_cd
        <uint16> <string>         <int64> <string> <date32>   <int64> <string>  <string>                    <int64> <string>
    0         0 CS021313000114          1 female   1981-04-29      37 259-1113  S14021                   20150905 0-00000000-0
    1         1 CS037613000071          9 unknown  1952-04-01      66 136-0076  S13037                   20150414 0-00000000-0
    2         2 CS031415000172          1 female   1976-10-04      42 151-0053  S13031                   20150529 D-20100325-C
    3         3 CS028811000001          1 female   1933-03-27      86 245-0016  S14028                   20160115 0-00000000-0
    :         : :                       : :        :                : :         :                             : :
21967     21967 CS029414000065          1 female   1970-10-19      48 279-0043  S12029                   20150313 F-20101028-F
21968     21968 CS012403000043          0 male     1972-12-16      46 231-0825  S14012                   20150406 0-00000000-0
21969     21969 CS033512000184          1 female   1964-06-05      54 245-0016  S14033                   20160206 0-00000000-0
21970     21970 CS009213000022          1 female   1996-08-16      22 154-0012  S13009                   20150424 0-00000000-0
```

Example: Split data (2/2)

Create a randomly selected index vector:

```
train_indeces = df_customer_selected[:index].sample(0.8)
```

Output

```
#<RedAmber::Vector(:uint16, size=17576):0x000000000000fe10>
[5912, 1998, 7974, 12093, 6585, 3801, 10037, 11205, 17626, 16713, 15059, 5382, ... ]
```

Select records by the vector:

```
train =
df_customer_selected.slice(train_indeces)
```

Output

```
#<RedAmber::DataFrame : 17576 x 10 Vectors>
          index customer_id    ...
        <uint16> <string>       ...
    0      5912 CS022714000035 ...
    1      1998 CS004414000271 ...
    2      7974 CS013415000059 ...
    3     12093 CS012413000059 ...
    :         : :                  :
17572      4536 CS008515000065 ...
17573     18654 CS015514000045 ...
17574     21098 CS004412000397 ...
17575     19041 CS032314000077 ...
```

Reject records by the vector:

```
test =
df_customer_selected.remove(train_indeces)
```

```
#<RedAmber::DataFrame : 4395 x 10 Vectors>
         index customer_id    ...
       <uint16> <string>       ...
   0         2 CS031415000172 ...
   1         9 CS033513000180 ...
   2        11 CS035614000014 ...
   3        13 CS009413000079 ...
   :         : :                  :
4391     21947 CS005313000401 ...
4392     21958 CS003715000199 ...
4393     21961 CS012415000309 ...
4394     21970 CS009213000022 ...
```

What I did in RedAmber - (2) Common selector

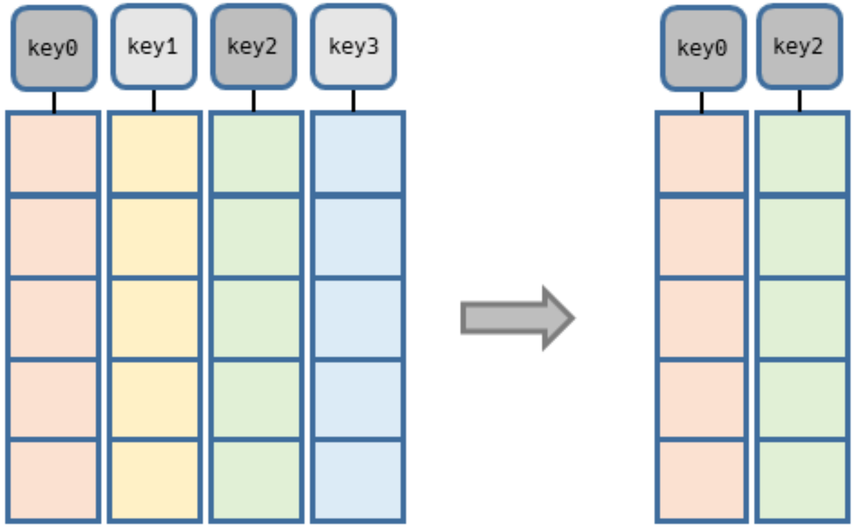
- Selectors for columns and rows;
  - Treat **boolean filter** and **index selector** equally as selector.

Column (variable) operations

DataFrame#**[]**(*keys*)

DataFrame#**pick**(*keys*)

DataFrame#**pick** { *keys* }

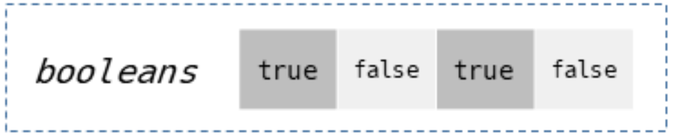


DataFrame#**pick**(*indices*)

DataFrame#**pick** { *indices* }

DataFrame#**pick**(*booleans*)

DataFrame#**pick** { *booleans* }



Also we have complementary brother;
DataFrame#**drop**

Row (observation) operations

```
DataFrame#[](indices)
DataFrame#slice(indices)
DataFrame#slice { indices }
```

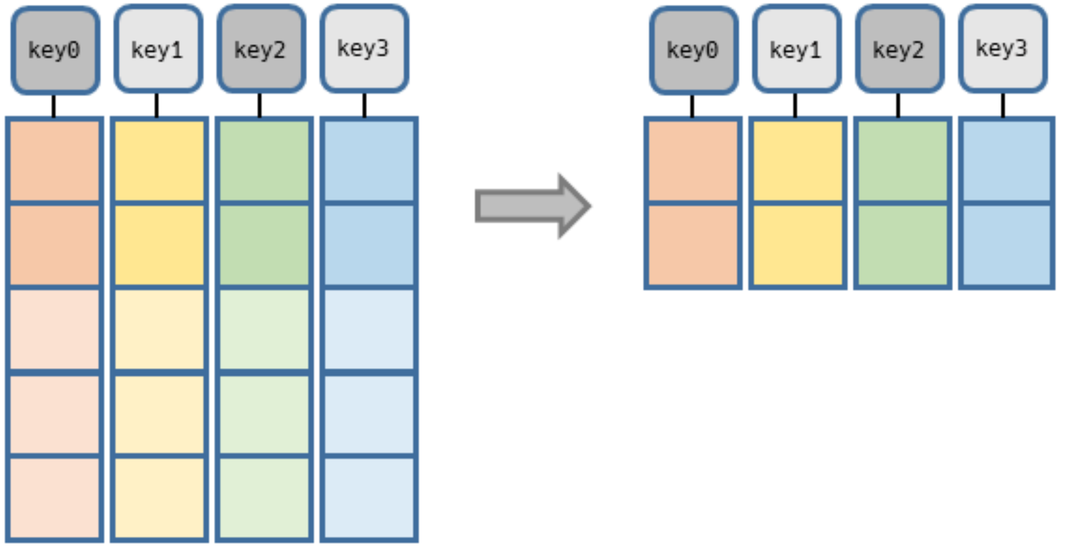
indices

0
1

```
DataFrame#[](booleans)
DataFrame#slice(booleans)
DataFrame#slice { booleans }
```

booleans

true
true
false
false
nil



Also we have complementary sister;
DataFrame#**remove**

Arrays, **Arrow::Array**s, **Vector**s are acceptable for *indices* and *Booleans*.

- Prepared #**filter** for the alias of #**slice** with booleans

What I did in RedAmber - (3) To use blocks effectively

- Reciever is self inside of block
  - Called by `BasicObject#instance_eval`
  - Column name will become method name by `method_missing`

```
# We can write:
dataframe.filter { amount > 1000 }

# Rather than:
dataframe.filter(dataframe.amount > 1000)

# Or
dataframe.filter(dataframe[:amount] > 1000)
```

Example: Select records

From #5 データサイエンス100本ノック(構造化データ加工編), Partially translated to alphabetical data.
Github: The-Japan-DataScientist-Society/100knocks-preprocess

df_receipt

```
#<RedAmber::DataFrame : 104681 x 9 Vectors>
       sales_ymd sales_epoch store_cd receipt_no receipt_sub_no customer_id    product_cd quantity  amount
         <int64>     <int64> <string>    <int64>        <int64> <string>       <string>    <int64> <int64>
     0  20181103  1541203200 S14006          112              1 CS006214000001 P070305012         1     158
     1  20181118  1542499200 S13008         1132              2 CS008415000097 P070701017         1      81
     2  20170712  1499817600 S14028         1102              1 CS028414000014 P060101005         1     170
   :         :           : :                 :              : :                :                :       :
104678  20170311  1489190400 S14040         1122              1 CS040513000195 P050405003         1     168
104679  20170331  1490918400 S13002         1142              1 CS002513000049 P060303001         1     148
104680  20190423  1555977600 S13016         1102              2 ZZ000000000000 P050601001         1     138
```

Code

```
df_receipt
  .pick(:sales_ymd, :customer_id, :product_cd, :amount)
  .slice { (customer_id == 'CS018205000001') & (amount >= 1000) }
```

Output

```
#<RedAmber::DataFrame : 3 x 4 Vectors>
  sales_ymd customer_id    product_cd  amount
    <int64> <string>       <string>   <int64>
0  20180911 CS018205000001 P071401012    2200
1  20190226 CS018205000001 P071401020    2200
2  20180911 CS018205000001 P071401005    1100
```

What I did in RedAmber - (4) TDR

- Table style is cramped for displaying many columns.

```
penguins
=>
#<RedAmber::DataFrame : 344 x 8 Vectors, 0x0000000000084fd0>
    species  island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g sex          year
    <string> <string>        <double>      <double>            <uint8>    <uint16> <string> <uint16>
  0 Adelie   Torgersen           39.1          18.7                181        3750 male         2007
  1 Adelie   Torgersen           39.5          17.4                186        3800 female       2007
  2 Adelie   Torgersen           40.3          18.0                195        3250 female       2007
  3 Adelie   Torgersen          (nil)         (nil)               (nil)       (nil) (nil)        2007
  4 Adelie   Torgersen           36.7          19.3                193        3450 female       2007
  5 Adelie   Torgersen           39.3          20.6                190        3650 male         2007
  : :        :                      :             :                  :           : :              :
340 Gentoo   Biscoe              46.8          14.3                215        4850 female       2009
341 Gentoo   Biscoe              50.4          15.7                222        5750 male         2009
342 Gentoo   Biscoe              45.2          14.8                212        5200 female       2009
343 Gentoo   Biscoe              49.9          16.1                213        5400 male         2009
```

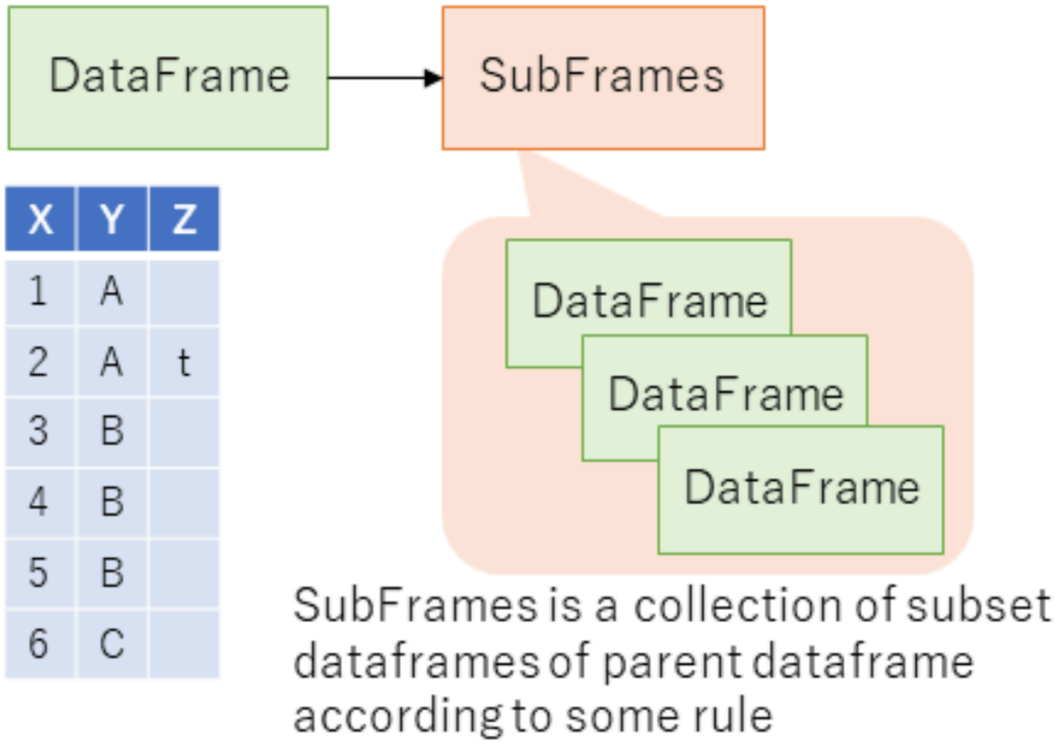
What I did in RedAmber - (4) TDR -cont'd

- **TDR** (**T**ransposed **D**ataFrame **R**epresentation)
  - Showing DataFrame in transposed style and provide summarized information.
  - Similar to `glimpse` in R or Polars, but more helpful and detailed.

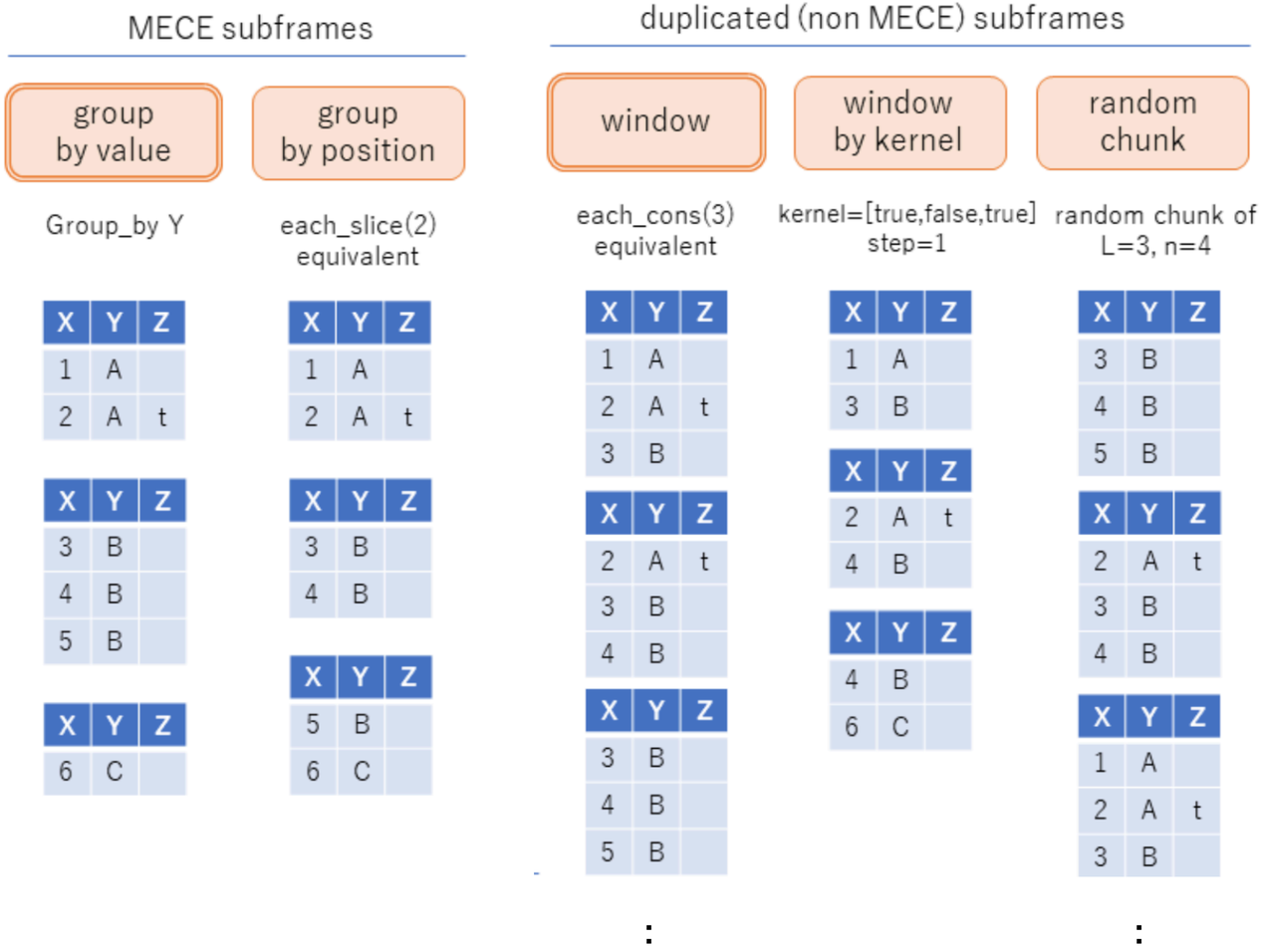
```
penguins.tdr
=>
RedAmber::DataFrame : 344 x 8 Vectors
Vectors : 5 numeric, 3 strings
# key                type   level data_preview
0 :species           string     3 {"Adelie"=>152, "Chinstrap"=>68, "Gentoo"=>124}
1 :island            string     3 {"Torgersen"=>52, "Biscoe"=>168, "Dream"=>124}
2 :bill_length_mm    double   165 [39.1, 39.5, 40.3, nil, 36.7, ... ], 2 nils
3 :bill_depth_mm     double    81 [18.7, 17.4, 18.0, nil, 19.3, ... ], 2 nils
4 :flipper_length_mm uint8     56 [181, 186, 195, nil, 193, ... ], 2 nils
5 :body_mass_g       uint16    95 [3750, 3800, 3250, nil, 3450, ... ], 2 nils
6 :sex               string     3 {"male"=>168, "female"=>165, nil=>11}
7 :year              uint16     3 {2007=>110, 2008=>114, 2009=>120}
```

What I did in RedAmber - (5) SubFrames

- **SubFrames** is a class to represent ordered subsets of a DataFrame.
  - Elements are also **DataFrame**s

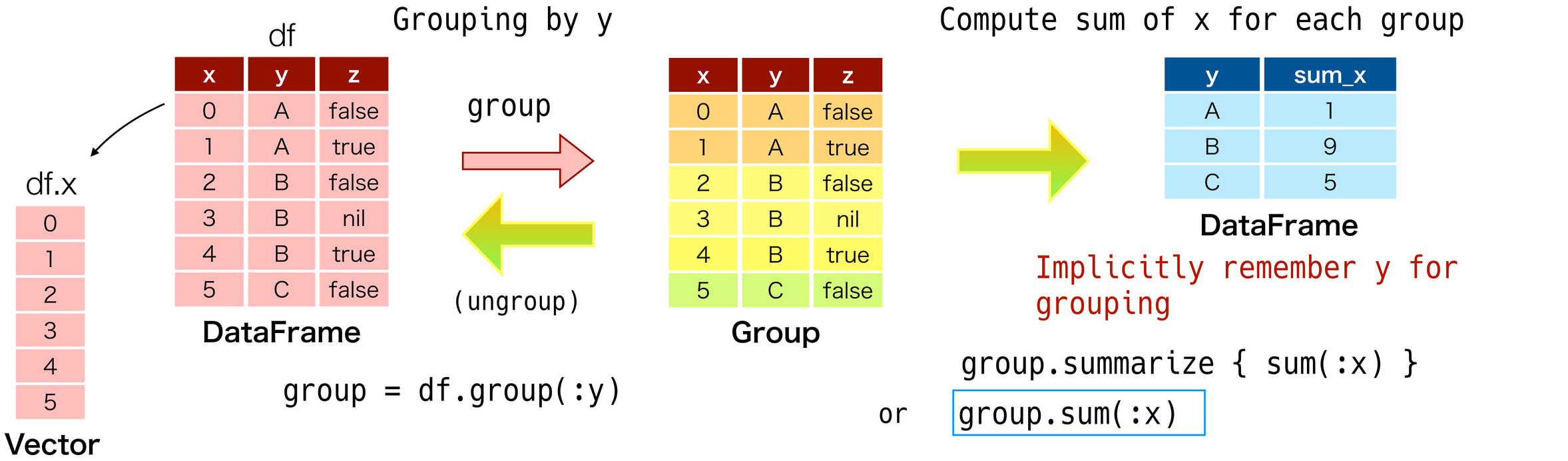


X	Y	Z
1	A	
2	A	t
3	B	
4	B	
5	B	
6	C	



Grouping in RedAmber

Compatible API in other languages/libraries



x	y	z
0	A	false
1	A	true
2	B	false
3	B	nil
4	B	true
5	C	false

df.x
0
1
2
3
4
5

y	sum_x
A	1
B	9
C	5

```
# Sum of Vector
df.x.sum  #=> 15
```

- It is functional style for #sum.
- different function is required.
  - Arrow have `hash_*` for group operation.

SubFrames is ordered subsets of a DataFrame

Grouping by y's value

df

x	y	z
0	A	false
1	A	true
2	B	false
3	B	nil
4	B	true
5	C	false

DataFrame

```
sf =
df.sub_group(:y)
```

API by class Group

```
group = df.group(:y)
```

sf

x	y	z
0	A	false
1	A	true

DataFrame

x	y	z
2	B	false
3	B	nil
4	B	true

DataFrame

x	y	z
5	C	false

DataFrame

SubFrames

```
For each DataFrame;
  Value of y,
  Sum of x.
```

y	sum_x
A	1
B	9
C	5

DataFrame

```
sf.aggregate do
  { y: y.first, sum_x: x.sum }
end
```

```
group.summarize { sum(:x) }
```

```
group.sum(:x)
```

Other way to create SubFrames

Windowing by position

df

window with size: 3

x	y	z
0	A	false
1	A	true
2	B	false
3	B	nil
4	B	true
5	C	false

DataFrame

sf

x	y	z
0	A	false
1	A	true
2	B	false

DataFrame

x	y	z
1	A	true
2	B	false
3	B	nil

DataFrame

x	y	z
2	B	false
3	B	nil
4	B	true

DataFrame

x	y	z
3	B	nil
4	B	true
5	C	false

DataFrame

SubFrames

```
sf = df.sub_by_window(size: 3)

Options:
  - from: 0
  - step: 1
```

```
For each DataFrame;
  compute mean x,
  count non-nil elements in z.
```

mean_x	count_z
1	3
2	2
3	2
4	2

DataFrame

```
sf.aggregate do
  { mean_x: x.mean, count_z: z.count }
end
```

SubFrames: Element-wise operation

df

x	y	z
0	A	false
1	A	true
2	B	false
3	B	nil
4	B	true
5	C	false

DataFrame

Grouping by y to create collection of DataFrames

```
sf =

df.sub_by_value(keys: :y)
```

sf

x	y	z
0	A	false
1	A	true

DataFrame

x	y	z
2	B	false
3	B	nil
4	B	true

DataFrame

x	y	z
5	C	false

DataFrame

SubFrames

For each DataFrame;
Update x by the index from 1,
Create cumsum c.

```
sf.assign do
  {
    x: indices(1),
    c: x.cumsum,
  }
end
```

sf

x	y	z	c
1	A	false	0
2	A	true	1

DataFrame

x	y	z	c
1	B	false	2
2	B	nil	5
3	B	true	9

DataFrame

x	y	z	c
1	C	false	5

DataFrame

SubFrames

Operation of a **DataFrame**

```
df.assign do
  {
    x: indices(1),
    c: x.cumsum,
  }
end
```

x	y	z	c
1	A	false	0
2	A	true	1
3	B	false	3
4	B	nil	6
5	B	true	10
6	C	false	15

DataFrame

Operations to get DataFrame from SubFrames

sf

x	y	z	c
1	A	false	0
2	A	true	1

DataFrame

x	y	z	c
1	B	false	2
2	B	nil	5
3	B	true	9

DataFrame

x	y	z	c
1	C	false	5

DataFrame

SubFrames

Concatenation

```
sf.concatenate
```

x	y	z	c
1	A	false	0
2	A	true	1
1	B	false	2
2	B	nil	5
3	B	true	9
1	C	false	5

DataFrame

Using SubFrames are examples of **“split-apply-combine"** strategy, well suited to Ruby!

Detection

```
sf.find { |df| df.c.max > 8 }
```

x	y	z	c
1	B	false	2
2	B	nil	5
3	B	true	9

DataFrame

Aggregation

```
sf.aggregate do
  { x: x.last, y: y.first, c: c.max }
end
```

x	y	c
2	A	1
3	B	9
1	C	5

DataFrame

Example: RubyKaigi (1/6)

Code

```
# load from here document as csv
rubykaigi = DataFrame.load(Arrow::Buffer.new(<<~CSV), format: :csv)
  year,city,venue,venue_en
  2015,東京都中央区,ベルサール汐留,"Bellesalle Shiodome"
  2016,京都府京都市左京区,京都国際会議場,"Kyoto International Conference Center"
  2017,広島県広島市中区,広島国際会議場,"International Conference Center Hiroshima"
  2018,宮城県仙台市青葉区,仙台国際センター,"Sendai International Center"
  2019,福岡県福岡市博多区,福岡国際会議場,"Fukuoka International Congress Center"
  2022,三重県津市,三重県総合文化センター,"Mie Center for the Arts"
  2023,長野県松本市,松本市民芸術館,"Matsumoto Performing Arts Centre"
CSV
```

Output

```
#<RedAmber::DataFrame : 7 x 4 Vectors>
     year city                venue                  venue_en
  <int64> <string>            <string>               <string>
0    2015 東京都中央区        ベルサール汐留         Bellesalle Shiodome
1    2016 京都府京都市左京区  京都国際会議場         Kyoto International Conference Center
2    2017 広島県広島市中区    広島国際会議場         International Conference Center Hiroshima
3    2018 宮城県仙台市青葉区  仙台国際センター       Sendai International Center
4    2019 福岡県福岡市博多区  福岡国際会議場         Fukuoka International Congress Center
5    2022 三重県津市          三重県総合文化センター Mie Center for the Arts
6    2023 長野県松本市        松本市民芸術館         Matsumoto Performing Arts Centre
```

Example: RubyKaigi (2/6)

Code

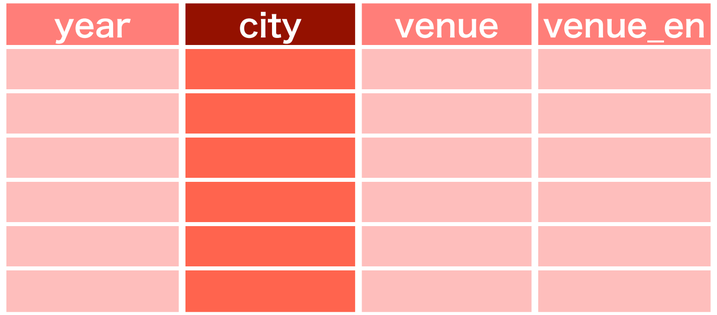
```
geo =
  DataFrame.new(Datasets::Geolonia.new)  # Read Geolonia data from Red Datasets.
    .drop(%w[prefecture_kana municipality_kana street_kana alias])
    .assign(:prefecture_romaji) { prefecture_romaji.map { _1.split[0].capitalize } }  # 'OSAKA FU' => 'Osaka'
    .assign(:municipality_romaji) do  # 'OSAKA SHI NANIWA KU' => 'Naniwa-ku, Osaka-shi'
      municipality_romaji
        .map do |city_string|
          cities = city_string.split.each_slice(2).to_a.reverse
          cities.map do |name, municipality|
            "#{name.capitalize}-#{municipality.downcase}"
          end.join(', ')
        end
    end
    .assign(:street_romaji) { street_romaji.map { _1.nil? ? nil : _1.capitalize } }
    .assign{ [:latitude, :longitude].map { |var| [var, v(var).cast(:double)] } }
    .rename(prefecture_name: :prefecture, municipality_name: :municipality, street_name: :street)
    .assign(:city) { prefecture.merge(municipality, sep: '') }
    .assign(:city_romaji) { municipality_romaji.merge(prefecture_romaji, sep: ', ') }
    .group(:city, :city_romaji)
    .summarize(:latitude, :longitude) { [mean(:latitude), mean(:longitude)] }  # set lat. and long. as its mean over municipality
```

Output

```
#<RedAmber::DataFrame : 1894 x 4 Vectors>
     city                 city_romaji                                 latitude longitude
     <string>             <string>                                    <double>  <double>
   0 北海道札幌市中央区   Chuo-ku, Sapporo-shi, Hokkaido                 43.05    141.34
   1 北海道札幌市北区     Kita-ku, Sapporo-shi, Hokkaido                 43.11    141.34
   2 北海道札幌市東区     Higashi-ku, Sapporo-shi, Hokkaido               43.1    141.37
   3 北海道札幌市白石区   Shiroishi-ku, Sapporo-shi, Hokkaido            43.05    141.41
   : :                    :                                                 :         :
```

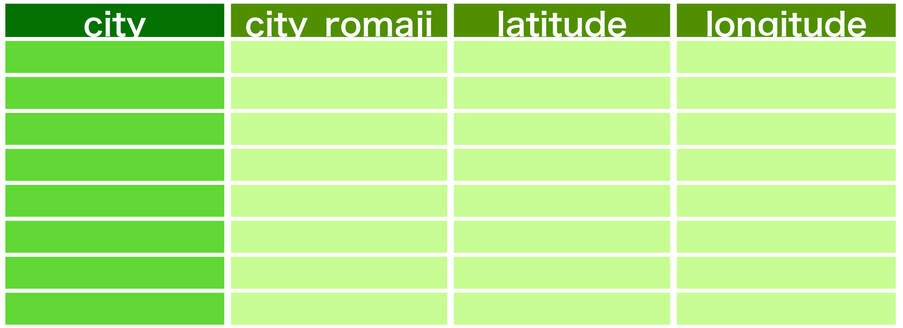
Example: RubyKaigi (3/6)

rubykaigi



DataFrame

geo



DataFrame

Code

```ruby
rubykaigi                # `left_join` will join matching values in left from right
  .left_join(geo)        # Join keys are automatically selected as `:city` (Natural join)
  .drop(:city, :venue)
```

Output

```
#<RedAmber::DataFrame : 7 x 5 Vectors>
     year venue_en                                 city_romaji                          latitude longitude
  <int64> <string>                                 <string>                             <double>  <double>
0    2015 Bellesalle Shiodome                      Chuo-ku, Tokyo                          35.68    139.78
1    2016 Kyoto International Conference Center    Sakyo-ku, Kyoto-shi, Kyoto              35.05    135.79
2    2017 International Conference Center Hiroshima Naka-ku, Hiroshima-shi, Hiroshima      34.38    132.45
3    2018 Sendai International Center              Aoba-ku, Sendai-shi, Miyagi             38.28     140.8
4    2019 Fukuoka International Congress Center    Hakata-ku, Fukuoka-shi, Fukuoka         33.58    130.44
5    2022 Mie Center for the Arts                  Tsu-shi, Mie                            34.71    136.46
6    2023 Matsumoto Performing Arts Centre         Matsumoto-shi, Nagano                   36.22    137.96
```

Example: RubyKaigi (4/6)

Code

```
rubykaigi_location =
  rubykaigi
    .left_join(geo)
    .pick(:latitude, :longitude)
    .assign_left(:location) { propagate('RubyKaigi') }
```

Output

```
#<RedAmber::DataFrame : 7 x 3 Vectors>
  location  latitude longitude
  <string>  <double>  <double>
0 RubyKaigi    35.68    139.78
1 RubyKaigi    35.05    135.79
2 RubyKaigi    34.38    132.45
3 RubyKaigi    38.28     140.8
4 RubyKaigi    33.58    130.44
5 RubyKaigi    34.71    136.46
6 RubyKaigi    36.22    137.96
```

```
cities_all =
  geo
    .pick(:latitude, :longitude)
    .assign_left(:location) { propagate('Japan') }
```

```
#<RedAmber::DataFrame : 1894 x 3 Vectors>
     location latitude longitude
     <string> <double>  <double>
   0 Japan        43.05    141.34
   1 Japan        43.11    141.34
   2 Japan         43.1    141.37
   3 Japan        43.05    141.41
   4 Japan        43.03    141.38
   5 Japan        42.98    141.32
   : :               :         :
```

Code

```
locations = rubykaigi_location.concatenate(cities_all)
locations.group(:location)
```

Output

```
#<RedAmber::Group : 0x000000000000fec4>
  location  group_count
  <string>      <int64>
0 RubyKaigi           7
1 Japan            1894
```

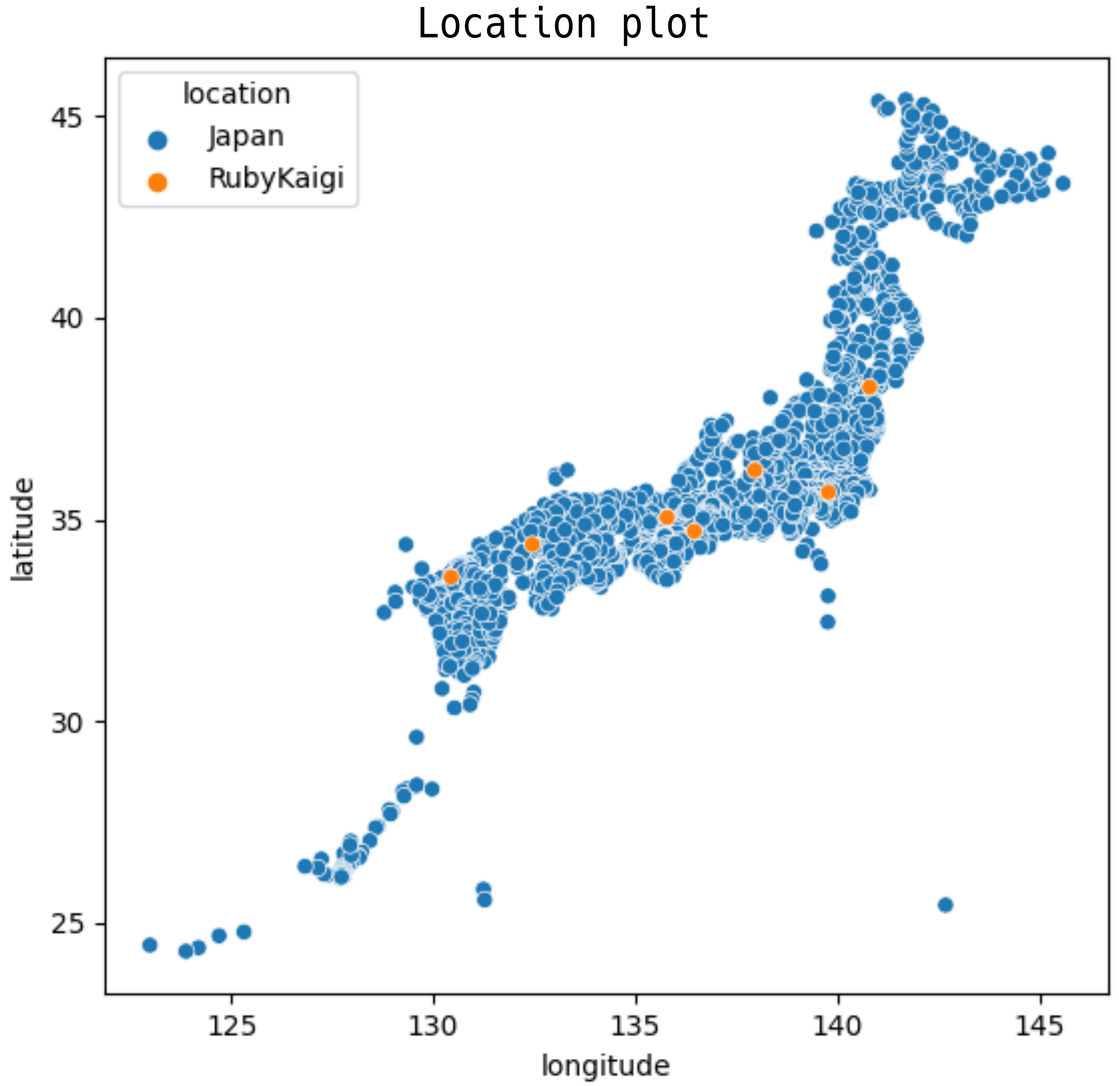
Example: RubyKaigi (5/6)

`locations`

```
   #<RedAmber::DataFrame : 1901 x 3
Vectors>
     location  latitude longitude
     <string>  <double>  <double>
   0 RubyKaigi    35.68    139.78
   1 RubyKaigi    35.05    135.79
   2 RubyKaigi    34.38    132.45
   3 RubyKaigi    38.28     140.8
   : :               :         :
1897 Japan         26.14    127.73
1898 Japan         24.69     124.7
1899 Japan          24.3    123.88
1900 Japan         24.46    122.99
```

Code

```
require 'charty'
Charty::Backends.use(:pyplot)

Charty.scatter_plot(
  data:  locations.table,
  x:     :longitude,
  y:     :latitude,
  color: :location
)
```



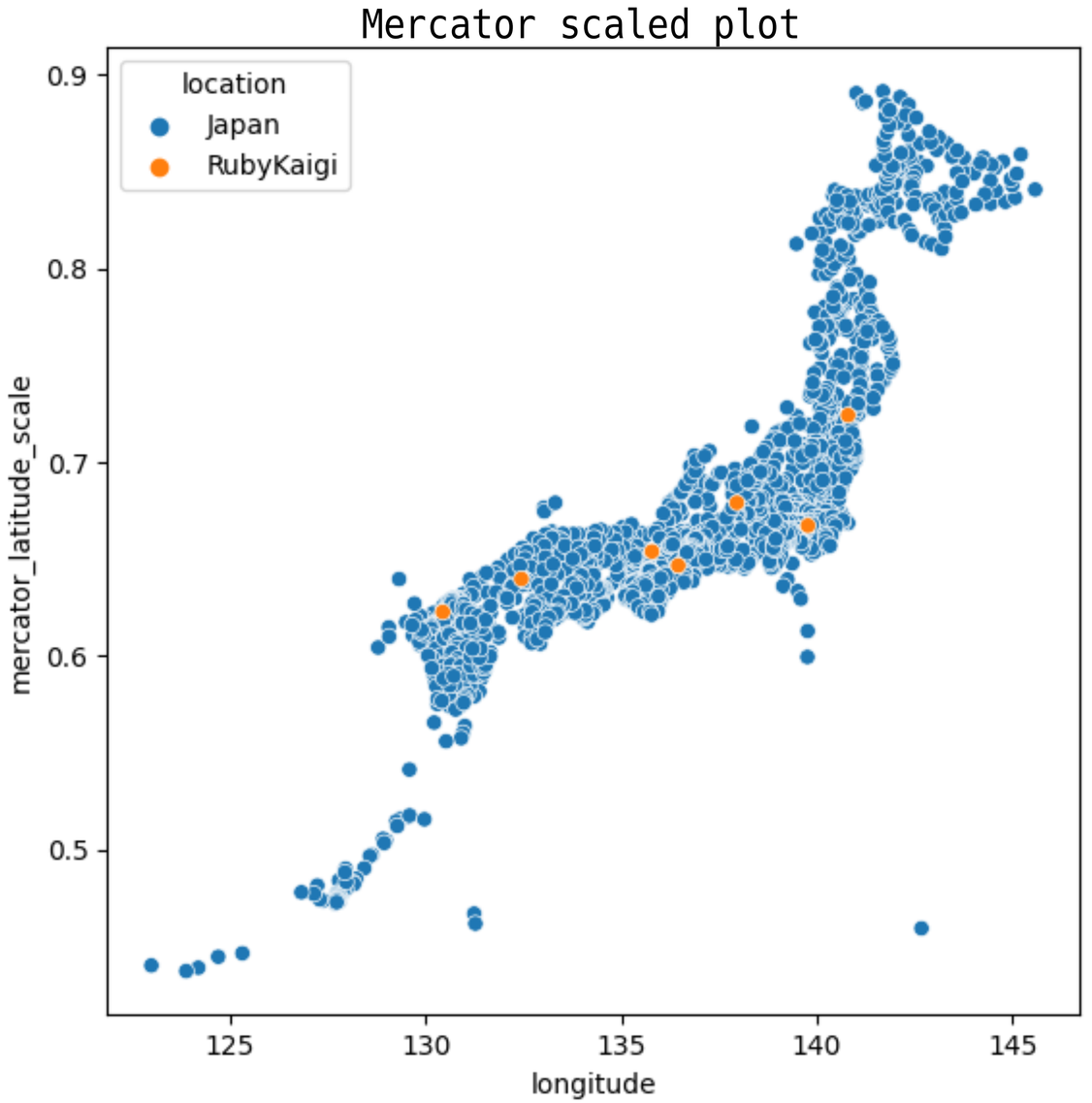
Example: RubyKaigi (6/6)

Code

```
mercator =
  locations
    .assign(:mercator_latitude_scale) do
      scales = (Math::PI * (latitude + 90) / 360).tan.ln
    end
Charty.scatter_plot(
  data:  mercator.table,
  x:     :longitude,
  y:     :mercator_latitude_scale,
  color: :location
)
```

`mercator`

```
#<RedAmber::DataFrame : 1901 x 4 Vectors>
     location  latitude longitude mercator_latitude_scale
     <string>  <double>  <double>                  <double>
   0 RubyKaigi    35.68    139.78                      0.67
   1 RubyKaigi    35.05    135.79                      0.65
   2 RubyKaigi    34.38    132.45                      0.64
   3 RubyKaigi    38.28     140.8                      0.72
   : :               :         :
1897 Japan        26.14    127.73                      0.47
1898 Japan        24.69     124.7                      0.44
1899 Japan         24.3    123.88                      0.44
1900 Japan        24.46    122.99                      0.44
```



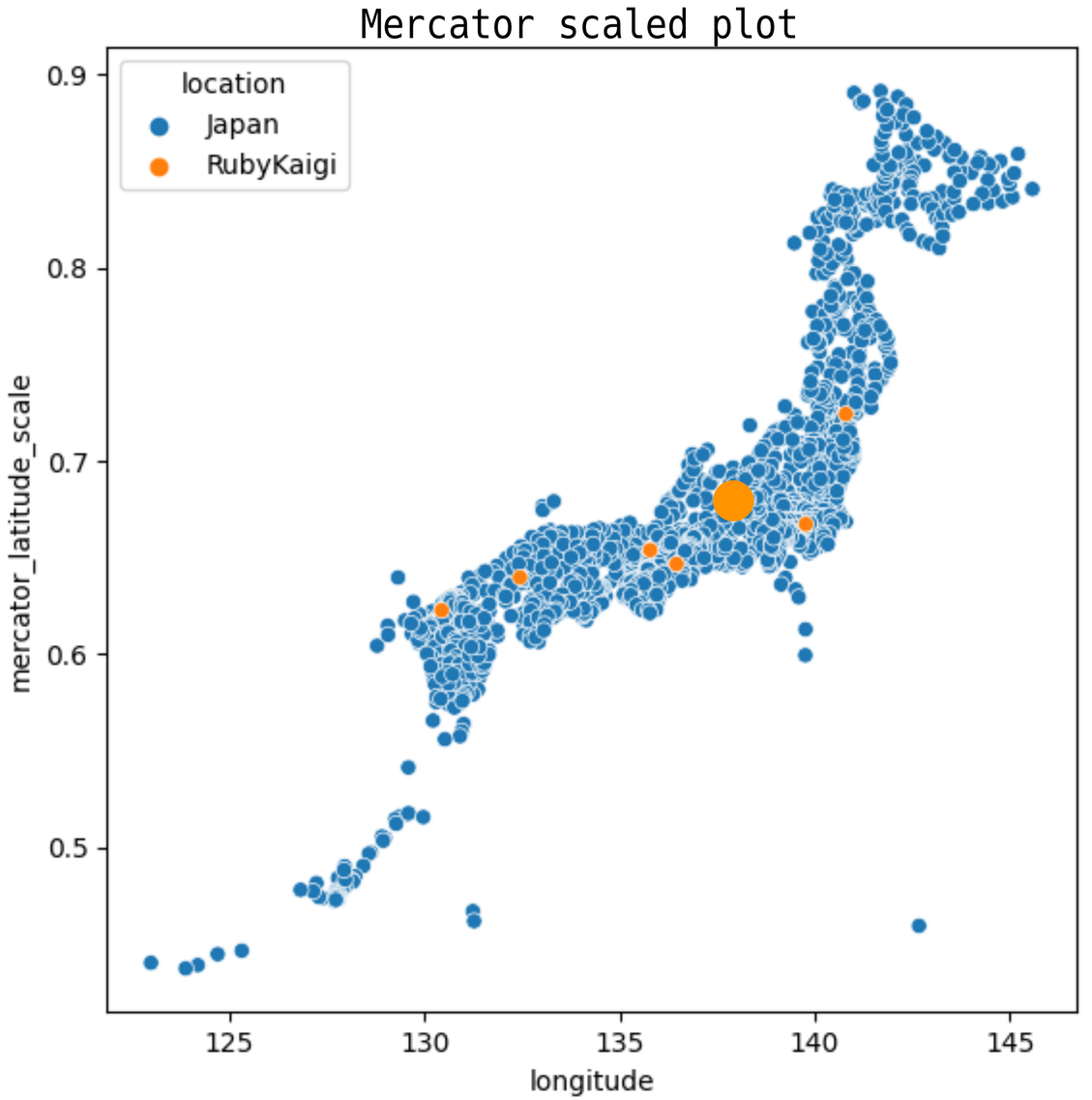
Example: RubyKaigi (6/6)

Code

```
mercator =
  locations
    .assign(:mercator_latitude_scale) do
      scales = (Math::PI * (latitude + 90) / 360).tan.ln
    end
Charty.scatter_plot(
  data:  mercator.table,
  x:     :longitude,
  y:     :mercator_latitude_scale,
  color: :location
)
```

`mercator`

```
#<RedAmber::DataFrame : 1901 x 4 Vectors>
     location  latitude longitude mercator_latitude_scale
     <string>  <double>  <double>                  <double>
   0 RubyKaigi    35.68    139.78                      0.67
   1 RubyKaigi    35.05    135.79                      0.65
   2 RubyKaigi    34.38    132.45                      0.64
   3 RubyKaigi    38.28     140.8                      0.72
   : :               :         :
1897 Japan        26.14    127.73                      0.47
1898 Japan        24.69     124.7                      0.44
1899 Japan         24.3    123.88                      0.44
1900 Japan        24.46    122.99                      0.44
```



Destination of the Adventure

- I was a beginner in OSS development, but a year-long adventure led me here in **Matsumoto** !
- **RedAmber**
  - is a library designed to provide idiomatic Ruby interface.
  - is a DataFrame for Rubyists (I hope).
- **Ruby** has great potential in the field of data processing.
  - Because data processing with Ruby is confortable and fun !

Where to go next?

- More examples
  - I will complete `100knocks-preprocess` in RedAmber.
- Faster impremementation
  - In RedAmber itself
  - Translate the workload (query) to another engine
  - Contribute to Substrait

Thanks

- My mentor of Ruby Association Grant, Kenta Murata (@mrkn) for his kindful help.
- Sutou Kouhei (@kou) for his wide-ranging advice on Red Arrow commits and RedAmber bugs.
- Benson Muite (@bkmgit) added the Fedra testing workflow and the Julia section of the comparison table with other data frames.
- @kojix2 contributed to the code by adding the YARD documentation generation workflow and modifying the documentation.
- I would also like to thank the members of Red Data Tools Gitter for their valuable comments and suggestions.
- Ruby Association for giving me financial support.
- I would like to express my deepest gratitude to Matz and everyone in the Ruby community for creating and growing Ruby !

See you at Red Data Tools, thank you !

Powered by Rabbit 3.0.1