

# Proposed API for tech.ml.dataset

GenerateMe

2020-05-26

## Introduction

tech.ml.dataset is a great and fast library which brings columnar dataset to the Clojure. Chris Nuernberger has been working on this library for last year as a part of bigger **tech.ml** stack.

I've started to test the library and help to fix uncovered bugs. My main goal was to compare functionalities with the other standards from other platforms. I focused on R solutions: dplyr, tidyr and data.table.

During conversions of the examples I've come up how to reorganized existing **tech.ml.dataset** functions into simple to use API. The main goals were:

- Focus on dataset manipulation functionality, leaving other parts of **tech.ml** like pipelines, datatypes, readers, ML, etc.
- Single entry point for common operations - one function dispatching on given arguments.
- **group-by** results with special kind of dataset - a dataset containing subsets created after grouping as a column.
- Most operations recognize regular dataset and grouped dataset and process data accordingly.
- One function form to enable thread-first on dataset.

All proposed functions are grouped in tabs below. Select group to see examples and details.

If you want to know more about **tech.ml.dataset** and **tech.ml.datatype** please refer their documentation:

- Datatype
- Date/time
- Dataset

INFO: The future of this API is not known yet. Two directions are possible: integration into **tech.ml** or development under Scicloj organization. For the time being use this repo if you want to try. Join the discussion on Zulip

Let's require main namespace and define dataset used in most examples:

```
(require '[techtest.api :as api])
(def DS (api/dataset {:V1 (take 9 (cycle [1 2]))
                     :V2 (range 1 10)
                     :V3 (take 9 (cycle [0.5 1.0 1.5]))
                     :V4 (take 9 (cycle ["A" "B" "C"]))}))
```

DS

\_\_unnamed [9 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	2	1.000	B
1	3	1.500	C

:V1	:V2	:V3	:V4
2	4	0.5000	A
1	5	1.000	B
2	6	1.500	C
1	7	0.5000	A
2	8	1.000	B
1	9	1.500	C

## Functionality

### Dataset

Dataset is a special type which can be considered as a map of columns implemented around `tech.ml.datatype` library. Each column can be considered as named sequence of typed data. Supported types include integers, floats, string, boolean, date/time, objects etc.

### Dataset creation

Dataset can be created from various of types of Clojure structures and files:

- single values
- sequence of maps
- map of sequences or values
- sequence of columns (taken from other dataset or created manually)
- sequence of pairs
- file types: raw/gzipped csv/tsv, json, xls(x) taken from local file system or URL
- input stream

`api/dataset` accepts:

- data
- options (see documentation of `tech.ml.dataset/->dataset` function for full list):
- `:dataset-name` - name of the dataset
- `:num-rows` - number of rows to read from file
- `:header-row?` - indication if first row in file is a header
- `:key-fn` - function applied to column names (eg. `keyword`, to convert column names to keywords)
- `:separator` - column separator
- `:single-value-column-name` - name of the column when single value is provided

---

Empty dataset.

```
(api/dataset)
```

```
_unnamed [0 0]
```

---

Dataset from single value.

```
(api/dataset 999)
```

```
_unnamed [1 1]:
```

---

:\$value
999

---



---

Set column name for single value. Also set the dataset name.

```
(api/dataset 999 {:$single-value-column-name "my-single-value"})
(api/dataset 999 {:$single-value-column-name ""
                  :dataset-name "Single value"})
```

\_\_unnamed [1 1]:

---

my-single-value
999

---

Single value [1 1]:

---

0
999

---



---

Sequence of pairs (first = column name, second = value(s)).

```
(api/dataset [[:A 33] [:B 5] [:C :a]])
```

\_\_unnamed [1 3]:

---

:A	:B	:C
33	5	:a

---



---

Not sequential values are repeated row-count number of times.

```
(api/dataset [[:A [1 2 3 4 5 6]] [:B "X"] [:C :a]])
```

\_\_unnamed [6 3]:

---

:A	:B	:C
1	X	:a
2	X	:a
3	X	:a
4	X	:a
5	X	:a
6	X	:a

---



---

Dataset created from map (keys = column name, second = value(s)). Works the same as sequence of pairs.

```
(api/dataset {:A 33})
(api/dataset {:A [1 2 3]})
(api/dataset {:A [3 4 5] :B "X"})
```

\_\_unnamed [1 1]:

:A
33

\_\_unnamed [3 1]:

:A
1
2
3

\_\_unnamed [3 2]:

:A	:B
3	X
4	X
5	X

---

You can put any value inside a column

```
(api/dataset {:A [[3 4 5] [:a :b]] :B "X"})
```

\_\_unnamed [2 2]:

:A	:B
[3 4 5]	X
[:a :b]	X

---

Sequence of maps

```
(api/dataset [{:a 1 :b 3} {:b 2 :a 99}])
(api/dataset [{:a 1 :b [1 2 3]} {:a 2 :b [3 4]}])
```

\_\_unnamed [2 2]:

:a	:b
1	3
99	2

\_\_unnamed [2 2]:

:a	:b
1	[1 2 3]
2	[3 4]

Missing values are marked by `nil`

```
(api/dataset [{:a nil :b 1} {:a 3 :b 4} {:a 11}])
```

\_\_unnamed [3 2]:

:a	:b
	1
3	4
11	

Import CSV file

```
(api/dataset "data/family.csv")
```

data/family.csv [5 5]:

family	dob_child1	dob_child2	gender_child1	gender_child2
1	1998-11-26	2000-01-29	1	2
2	1996-06-22		2	
3	2002-07-11	2004-04-05	2	2
4	2004-10-10	2009-08-27	1	1
5	2000-12-05	2005-02-28	2	1

Import from URL

```
(defonce ds (api/dataset "https://vega.github.io/vega-lite/examples/data/seattle-weather.csv"))
```

ds

https://vega.github.io/vega-lite/examples/data/seattle-weather.csv [1461 6]:

date	precipitation	temp_max	temp_min	wind	weather
2012-01-01	0.000	12.80	5.000	4.700	drizzle
2012-01-02	10.90	10.60	2.800	4.500	rain
2012-01-03	0.8000	11.70	7.200	2.300	rain
2012-01-04	20.30	12.20	5.600	4.700	rain
2012-01-05	1.300	8.900	2.800	6.100	rain
2012-01-06	2.500	4.400	2.200	2.200	rain
2012-01-07	0.000	7.200	2.800	2.300	rain
2012-01-08	0.000	10.00	2.800	2.000	sun
2012-01-09	4.300	9.400	5.000	3.400	rain
2012-01-10	1.000	6.100	0.6000	3.400	rain
2012-01-11	0.000	6.100	-1.100	5.100	sun

date	precipitation	temp_max	temp_min	wind	weather
2012-01-12	0.000	6.100	-1.700	1.900	sun
2012-01-13	0.000	5.000	-2.800	1.300	sun
2012-01-14	4.100	4.400	0.6000	5.300	snow
2012-01-15	5.300	1.100	-3.300	3.200	snow
2012-01-16	2.500	1.700	-2.800	5.000	snow
2012-01-17	8.100	3.300	0.000	5.600	snow
2012-01-18	19.80	0.000	-2.800	5.000	snow
2012-01-19	15.20	-1.100	-2.800	1.600	snow
2012-01-20	13.50	7.200	-1.100	2.300	snow
2012-01-21	3.000	8.300	3.300	8.200	rain
2012-01-22	6.100	6.700	2.200	4.800	rain
2012-01-23	0.000	8.300	1.100	3.600	rain
2012-01-24	8.600	10.00	2.200	5.100	rain
2012-01-25	8.100	8.900	4.400	5.400	rain

## Saving

Export dataset to a file or output stream can be done by calling `api/write-csv!`. Function accepts:

- dataset
- file name with one of the extensions: `.csv`, `.tsv`, `.csv.gz` and `.tsv.gz` or output stream
- options:
- `:separator` - string or separator char.

```
(api/write-csv! ds "output.tsv.gz")
(.exists (clojure.java.io/file "output.csv.gz"))
```

```
nil
true
```

## Dataset related functions

Summary functions about the dataset like number of rows, columns and basic stats.

---

Number of rows

```
(api/row-count ds)
```

```
1461
```

---

Number of columns

```
(api/column-count ds)
```

```
6
```

---

Shape of the dataset, [row count, column count]

```
(api/shape ds)
```

```
[1461 6]
```

---

General info about dataset. There are three variants:

- default - containing information about columns with basic statistics
- `:basic` - just name, row and column count and information if dataset is a result of **group-by** operation
- `:columns` - columns' metadata

```
(api/info ds)
(api/info ds :basic)
(api/info ds :columns)
```

<https://vega.github.io/vega-lite/examples/data/seattle-weather.csv>: descriptive-stats [6 10]:

:col-name	:datatype	:n-valid	:n-missing	:min	:mean	:mode	:max	:standard-deviation	:skew
date	:packed-local-date	1461	0	2012-01-01	2013-12-31	2015-12-31			
precipitation	:float32	1461	0	0.000	3.029	55.90	6.680	3.506	
temp_max	:float32	1461	0	-1.600	16.44	35.60	7.350	0.2809	
temp_min	:float32	1461	0	-7.100	8.235	18.30	5.023	-	0.2495
weather	:string	1461	0			sun			
wind	:float32	1461	0	0.4000	3.241	9.500	1.438	0.8917	

<https://vega.github.io/vega-lite/examples/data/seattle-weather.csv> :basic info [1 4]:

:name	:grouped?	:rows	:columns
<a href="https://vega.github.io/vega-lite/examples/data/seattle-weather.csv">https://vega.github.io/vega-lite/examples/data/seattle-weather.csv</a>	false	1461	6

<https://vega.github.io/vega-lite/examples/data/seattle-weather.csv> :column info [6 4]:

:name	:size	:datatype	:categorical?
date	1461	:packed-local-date	
precipitation	1461	:float32	
temp_max	1461	:float32	
temp_min	1461	:float32	
wind	1461	:float32	
weather	1461	:string	true

---

Getting a dataset name

```
(api/dataset-name ds)
```

"<https://vega.github.io/vega-lite/examples/data/seattle-weather.csv>"

---

Setting a dataset name (operation is immutable).

```
(->> "seattle-weather"
  (api/set-dataset-name ds)
  (api/dataset-name))
```

"seattle-weather"

## Columns and rows

Get columns and rows as sequences. `column`, `columns` and `rows` treat grouped dataset as regular one. See [Groups](#) to read more about grouped datasets.

---

Select column.

```
(ds "wind")
(api/column ds "date")
```

```
#tech.ml.dataset.column<float32>[1461]
wind
[4.700, 4.500, 2.300, 4.700, 6.100, 2.200, 2.300, 2.000, 3.400, 3.400, 5.100, 1.900, 1.300, 5.300, 3.200, ...]
#tech.ml.dataset.column<packed-local-date>[1461]
date
[2012-01-01, 2012-01-02, 2012-01-03, 2012-01-04, 2012-01-05, 2012-01-06, 2012-01-07, 2012-01-08, 2012-01-09, ...]
```

---

Columns as sequence

```
(take 2 (api/columns ds))
```

```
(#tech.ml.dataset.column<packed-local-date>[1461]
date
[2012-01-01, 2012-01-02, 2012-01-03, 2012-01-04, 2012-01-05, 2012-01-06, 2012-01-07, 2012-01-08, 2012-01-09, ...]
precipitation
[0.000, 10.90, 0.8000, 20.30, 1.300, 2.500, 0.000, 0.000, 4.300, 1.000, 0.000, 0.000, 0.000, 4.100, 5.300, ...])
```

---

Columns as map

```
(keys (api/columns ds :as-map))
```

```
("date" "precipitation" "temp_max" "temp_min" "wind" "weather")
```

---

Rows as sequence of sequences

```
(take 2 (api/rows ds))
```

```
([#object[java.time.LocalDate 0xfbb430e "2012-01-01"] 0.0 12.8 5.0 4.7 "drizzle"] [#object[java.time.LocalDate 0xfbb430e "2012-01-02"] 10.9 20.3 1.3 2.5 "drizzle"]])
```

---

Rows as sequence of maps

```
(clojure.pprint/pprint (take 2 (api/rows ds :as-maps)))
```

```
({"date" #object[java.time.LocalDate 0x7ea89b4a "2012-01-01"],
 "precipitation" 0.0,
 "temp_min" 5.0,
```



```

"weather" "drizzle",
"temp_max" 12.8,
"wind" 4.7}
{"date" #object[java.time.LocalDate 0x2682ae81 "2012-01-02"],
"precipitation" 10.9,
"temp_min" 2.8,
"weather" "rain",
"temp_max" 10.6,
"wind" 4.5})

```

## Printing

Dataset is printed using `dataset->str` or `print-dataset` functions. Options are the same as in `tech.ml.dataset/dataset-data->str`. Most important is `:print-line-policy` which can be one of the: `:single`, `:repl` or `:markdown`.

```
(api/print-dataset (api/group-by DS :V1) {:print-line-policy :markdown})
```

```
_unnamed [2 3]:
```

```

| :name | :group-id |
|-----|-----|
|      1 |          0 | Group: 1 [5 4]:<br><br>\| :V1 \| :V2 \|      :V3 \| :V4 \|<br>\|-----\|-----\|-----
|      2 |          1 |                                Group: 2 [4 4]:<br><br>\| :V1 \| :V2 \|      :

```

```
(api/print-dataset (api/group-by DS :V1) {:print-line-policy :repl})
```

```
_unnamed [2 3]:
```

```

| :name | :group-id | :data |
|-----|-----|-----|
|      1 |          0 | Group: 1 [5 4]:
|      |          |
|      |          | \| :V1 \| :V2 \|      :V3 \| :V4 \|
|      |          | \|-----\|-----\|-----\|-----\|
|      |          | \|      1 \|      1 \| 0.5000 \|      A \|
|      |          | \|      1 \|      3 \| 1.500 \|      C \|
|      |          | \|      1 \|      5 \| 1.000 \|      B \|
|      |          | \|      1 \|      7 \| 0.5000 \|      A \|
|      |          | \|      1 \|      9 \| 1.500 \|      C \|
|      2 |          1 | Group: 2 [4 4]:
|      |          |
|      |          | \| :V1 \| :V2 \|      :V3 \| :V4 \|
|      |          | \|-----\|-----\|-----\|-----\|
|      |          | \|      2 \|      2 \| 1.000 \|      B \|
|      |          | \|      2 \|      4 \| 0.5000 \|      A \|
|      |          | \|      2 \|      6 \| 1.500 \|      C \|
|      |          | \|      2 \|      8 \| 1.000 \|      B \|

```

## Group-by

Grouping by is an operation which splits dataset into subdatasets and pack it into new special type of... dataset. I distinguish two types of dataset: regular dataset and grouped dataset. The latter is the result of grouping.

Grouped dataset is annotated in by `:grouped?` meta tag and consist following columns:

- `:name` - group name or structure
- `:group-id` - integer assigned to the group
- `:data` - groups as datasets

Almost all functions recognize type of the dataset (grouped or not) and operate accordingly.

You can't apply reshaping or join/concat functions on grouped datasets.

## Grouping

Grouping is done by calling `group-by` function with arguments:

- `ds` - dataset
- `grouping-selector` - what to use for grouping
- options:
  - `:result-type` - what to return:
    - \* `:as-dataset` (default) - return grouped dataset
    - \* `:as-indexes` - return rows ids (row number from original dataset)
    - \* `:as-map` - return map with group names as keys and subdataset as values
    - \* `:as-seq` - return sequens of subdatasets
  - `:limit-columns` - list of the columns which should be returned during grouping by function.

All subdatasets (groups) have set name as the group name, additionally `group-id` is in meta.

Grouping can be done by:

- single column name
- seq of column names
- map of keys (group names) and row indexes
- value returned by function taking row as map

Note: currently dataset inside dataset is printed recursively so it renders poorly from markdown. So I will use `:as-seq` result type to show just group names and groups.

---

List of columns in groupd dataset

```
(api/column-names (api/group-by DS :V1))
```

```
(:name :group-id :data)
```

---

Content of the grouped dataset

```
(api/columns (api/group-by DS :V1) :as-map)
```

```
{:name #tech.ml.dataset.column<int64>[2]
:name
[1, 2, ], :group-id #tech.ml.dataset.column<int64>[2]
:group-id
[0, 1, ], :data #tech.ml.dataset.column<object>[2]
:data
[Group: 1 [5 4]:
```

```
| :V1 | :V2 | :V3 | :V4 |
|-----|-----|-----|-----|
| 1 | 1 | 0.5000 | A |
```

```

| 1 | 3 | 1.500 | C |
| 1 | 5 | 1.000 | B |
| 1 | 7 | 0.5000 | A |
| 1 | 9 | 1.500 | C |
, Group: 2 [4 4]:

| :V1 | :V2 | :V3 | :V4 |
|-----|-----|-----|-----|
| 2 | 2 | 1.000 | B |
| 2 | 4 | 0.5000 | A |
| 2 | 6 | 1.500 | C |
| 2 | 8 | 1.000 | B |
, ]}

```

---

Grouped dataset as map

```
(keys (api/group-by DS :V1 {:result-type :as-map}))
```

```
(1 2)
```

```
(vals (api/group-by DS :V1 {:result-type :as-map}))
```

```
(Group: 1 [5 4]:
```

:V1	:V2	:V3	:V4
1	1	0.5000	A
1	3	1.500	C
1	5	1.000	B
1	7	0.5000	A
1	9	1.500	C

```
Group: 2 [4 4]:
```

:V1	:V2	:V3	:V4
2	2	1.000	B
2	4	0.5000	A
2	6	1.500	C
2	8	1.000	B

```
)
```

---

Group dataset as map of indexes (row ids)

```
(api/group-by DS :V1 {:result-type :as-indexes})
```

```
{1 [0 2 4 6 8], 2 [1 3 5 7]}
```

---

Grouped datasets are printed as follows by default.

```
(api/group-by DS :V1)
```

\_\_unnamed [2 3]:

:name	:group-id	:data
1	0	Group: 1 [5 4]:
2	1	Group: 2 [4 4]:

To get groups as sequence or a map can be done from grouped dataset using `groups->seq` and `groups->map` functions.

Groups as seq can be obtained by just accessing `:data` column.

I will use temporary dataset here.

```
(let [ds (-> {"a" [1 1 2 2]
             "b" ["a" "b" "c" "d"]}
  (api/dataset)
  (api/group-by "a"))]
  (seq (ds :data))) ;; seq is not necessary but Markdown treats `:data` as command here
```

(Group: 1 [2 2]:

a	b
1	a
1	b

Group: 2 [2 2]:

a	b
2	c
2	d

)

```
(-> {"a" [1 1 2 2]
    "b" ["a" "b" "c" "d"]}
  (api/dataset)
  (api/group-by "a")
  (api/groups->seq))
```

(Group: 1 [2 2]:

a	b
1	a
1	b

Group: 2 [2 2]:

a	b
2	c
2	d

)

Groups as map

```
(-> {"a" [1 1 2 2]
     "b" ["a" "b" "c" "d"]})
(api/dataset)
(api/group-by "a")
(api/groups->map))
```

{1 Group: 1 [2 2]:

a	b
1	a
1	b

, 2 Group: 2 [2 2]:

a	b
2	c
2	d

}

Grouping by more than one column. You can see that group names are maps. When ungrouping is done these maps are used to restore column names.

```
(api/group-by DS [:V1 :V3] {:result-type :as-seq}))
```

(Group: {:V3 1.0, :V1 1} [1 4]:

:V1	:V2	:V3	:V4
1	5	1.000	B

Group: {:V3 0.5, :V1 1} [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
1	7	0.5000	A

Group: {:V3 0.5, :V1 2} [1 4]:

:V1	:V2	:V3	:V4
2	4	0.5000	A

Group: {:V3 1.0, :V1 2} [2 4]:

:V1	:V2	:V3	:V4
2	2	1.000	B
2	8	1.000	B

Group: {:V3 1.5, :V1 1} [2 4]:

:V1	:V2	:V3	:V4
1	3	1.500	C
1	9	1.500	C

Group: {:V3 1.5, :V1 2} [1 4]:

:V1	:V2	:V3	:V4
2	6	1.500	C

)

Grouping can be done by providing just row indexes. This way you can assign the same row to more than one group.

```
(api/group-by DS {"group-a" [1 2 1 2]
                  "group-b" [5 5 5 1]} {:result-type :as-seq})
```

(Group: group-a [4 4]:

:V1	:V2	:V3	:V4
2	2	1.000	B
1	3	1.500	C
2	2	1.000	B
1	3	1.500	C

Group: group-b [4 4]:

:V1	:V2	:V3	:V4
2	6	1.500	C
2	6	1.500	C
2	6	1.500	C
2	2	1.000	B

)

---

You can group by a result of grouping function which gets row as map and should return group name. When map is used as a group name, ungrouping restore original column names.

```
(api/group-by DS (fn [row] (* (:V1 row)
                              (:V3 row)))) {:result-type :as-seq})
```

(Group: 1.0 [2 4]:

:V1	:V2	:V3	:V4
2	4	0.5000	A
1	5	1.000	B

Group: 2.0 [2 4]:

:V1	:V2	:V3	:V4
2	2	1.000	B
2	8	1.000	B

Group: 0.5 [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
1	7	0.5000	A

Group: 3.0 [1 4]:

:V1	:V2	:V3	:V4
2	6	1.500	C

Group: 1.5 [2 4]:

:V1	:V2	:V3	:V4
1	3	1.500	C
1	9	1.500	C

)

---

You can use any predicate on column to split dataset into two groups.

```
(api/group-by DS (comp #(< % 1.0) :V3) {:result-type :as-seq}))
```

(Group: false [6 4]:

:V1	:V2	:V3	:V4
2	2	1.000	B
1	3	1.500	C
1	5	1.000	B
2	6	1.500	C
2	8	1.000	B
1	9	1.500	C

Group: true [3 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	4	0.5000	A
1	7	0.5000	A

)

juxt is also helpful

```
(api/group-by DS (juxt :V1 :V3) {:result-type :as-seq}))
```

(Group: [1 1.0] [1 4]:

:V1	:V2	:V3	:V4
1	5	1.000	B

Group: [1 0.5] [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
1	7	0.5000	A

Group: [2 1.5] [1 4]:

:V1	:V2	:V3	:V4
2	6	1.500	C

Group: [1 1.5] [2 4]:

:V1	:V2	:V3	:V4
1	3	1.500	C
1	9	1.500	C

Group: [2 0.5] [1 4]:



:V1	:V2	:V3	:V4
2	4	0.5000	A

Group: [2 1.0] [2 4]:

:V1	:V2	:V3	:V4
2	2	1.000	B
2	8	1.000	B

)

`tech.ml.dataset` provides an option to limit columns which are passed to grouping functions. It's done for performance purposes.

```
(api/group-by DS identity {:result-type :as-seq
                           :limit-columns [:V1]})
```

(Group: {:V1 1} [5 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
1	3	1.500	C
1	5	1.000	B
1	7	0.5000	A
1	9	1.500	C

Group: {:V1 2} [4 4]:

:V1	:V2	:V3	:V4
2	2	1.000	B
2	4	0.5000	A
2	6	1.500	C
2	8	1.000	B

)

## Ungrouping

Ungrouping simply concatenates all the groups into the dataset. Following options are possible

- `:order?` - order groups according to the group name ascending order. Default: `false`
- `:add-group-as-column` - should group name become a column? If yes column is created with provided name (or `:$group-name` if argument is `true`). Default: `nil`.
- `:add-group-id-as-column` - should group id become a column? If yes column is created with provided name (or `:$group-id` if argument is `true`). Default: `nil`.
- `:dataset-name` - to name resulting dataset. Default: `nil` (`_unnamed`)

If group name is a map, it will be splitted into separate columns. Be sure that groups (subdatasets) doesn't contain the same columns already.

If group name is a vector, it will be splitted into separate columns. If you want to name them, set vector of target column names as `:add-group-as-column` argument.

After ungrouping, order of the rows is kept within the groups but groups are ordered according to the internal storage.

---

Grouping and ungrouping.

```
(-> DS
  (api/group-by :V3)
  (api/ungroup))
```

\_\_unnamed [9 4]:

:V1	:V2	:V3	:V4
2	2	1.000	B
1	5	1.000	B
2	8	1.000	B
1	1	0.5000	A
2	4	0.5000	A
1	7	0.5000	A
1	3	1.500	C
2	6	1.500	C
1	9	1.500	C

---

Groups sorted by group name and named.

```
(-> DS
  (api/group-by :V3)
  (api/ungroup {:order? true
                 :dataset-name "Ordered by V3"}))
```

Ordered by V3 [9 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	4	0.5000	A
1	7	0.5000	A
2	2	1.000	B
1	5	1.000	B
2	8	1.000	B
1	3	1.500	C
2	6	1.500	C
1	9	1.500	C

---

Let's add group name and id as additional columns

```
(-> DS
  (api/group-by (comp #(< % 4) :V2))
  (api/ungroup {:add-group-as-column true
                :add-group-id-as-column true})))
```

\_\_unnamed [9 6]:

:\$group-name	:\$group-id	:V1	:V2	:V3	:V4
false	0	2	4	0.5000	A
false	0	1	5	1.000	B
false	0	2	6	1.500	C
false	0	1	7	0.5000	A
false	0	2	8	1.000	B
false	0	1	9	1.500	C
true	1	1	1	0.5000	A
true	1	2	2	1.000	B
true	1	1	3	1.500	C

Let's assign different column names

```
(-> DS
  (api/group-by (comp #(< % 4) :V2))
  (api/ungroup {:add-group-as-column "Is V2 less than 4?"
                :add-group-id-as-column "group id"})))
```

\_\_unnamed [9 6]:

Is V2 less than 4?	group id	:V1	:V2	:V3	:V4
false	0	2	4	0.5000	A
false	0	1	5	1.000	B
false	0	2	6	1.500	C
false	0	1	7	0.5000	A
false	0	2	8	1.000	B
false	0	1	9	1.500	C
true	1	1	1	0.5000	A
true	1	2	2	1.000	B
true	1	1	3	1.500	C

If we group by map, we can automatically create new columns out of group names.

```
(-> DS
  (api/group-by (fn [row] {"V1 and V3 multiplied" (* (:V1 row)
                                                       (:V3 row))
                           "V4 as lowercase" (clojure.string/lower-case (:V4 row))}))
  (api/ungroup {:add-group-as-column true})))
```

\_\_unnamed [9 6]:

V1 and V3 multiplied	V4 as lowercase	:V1	:V2	:V3	:V4
1.000	a	2	4	0.5000	A

V1 and V3 multiplied	V4 as lowercase	:V1	:V2	:V3	:V4
0.5000	a	1	1	0.5000	A
0.5000	a	1	7	0.5000	A
1.000	b	1	5	1.000	B
2.000	b	2	2	1.000	B
2.000	b	2	8	1.000	B
3.000	c	2	6	1.500	C
1.500	c	1	3	1.500	C
1.500	c	1	9	1.500	C

We can add group names without separation

```
(-> DS
  (api/group-by (fn [row] {"V1 and V3 multiplied" (* (:V1 row)
                                                    (:V3 row))
                        "V4 as lowercase" (clojure.string/lower-case (:V4 row))}))
  (api/ungroup {:add-group-as-column "just map"
               :separate? false}))
```

\_\_unnamed [9 5]:

just map		:V1	:V2	:V3	:V4
{“V1 and V3 multiplied” 1.0, “V4 as lowercase” “a”}		2	4	0.5000	A
{“V1 and V3 multiplied” 0.5, “V4 as lowercase” “a”}		1	1	0.5000	A
{“V1 and V3 multiplied” 0.5, “V4 as lowercase” “a”}		1	7	0.5000	A
{“V1 and V3 multiplied” 1.0, “V4 as lowercase” “b”}		1	5	1.000	B
{“V1 and V3 multiplied” 2.0, “V4 as lowercase” “b”}		2	2	1.000	B
{“V1 and V3 multiplied” 2.0, “V4 as lowercase” “b”}		2	8	1.000	B
{“V1 and V3 multiplied” 3.0, “V4 as lowercase” “c”}		2	6	1.500	C
{“V1 and V3 multiplied” 1.5, “V4 as lowercase” “c”}		1	3	1.500	C
{“V1 and V3 multiplied” 1.5, “V4 as lowercase” “c”}		1	9	1.500	C

The same applies to group names as sequences

```
(-> DS
  (api/group-by (juxt :V1 :V3))
  (api/ungroup {:add-group-as-column "abc"}))
```

\_\_unnamed [9 6]:

:abc-0	:abc-1	:V1	:V2	:V3	:V4
1	1.000	1	5	1.000	B
1	0.5000	1	1	0.5000	A
1	0.5000	1	7	0.5000	A
2	1.500	2	6	1.500	C
1	1.500	1	3	1.500	C
1	1.500	1	9	1.500	C
2	0.5000	2	4	0.5000	A
2	1.000	2	2	1.000	B
2	1.000	2	8	1.000	B

---

Let's provide column names

```
(-> DS
  (api/group-by (juxt :V1 :V3))
  (api/ungroup {:add-group-as-column ["v1" "v3"]}))
```

\_\_unnamed [9 6]:

v1	v3	:V1	:V2	:V3	:V4
1	1.000	1	5	1.000	B
1	0.5000	1	1	0.5000	A
1	0.5000	1	7	0.5000	A
2	1.500	2	6	1.500	C
1	1.500	1	3	1.500	C
1	1.500	1	9	1.500	C
2	0.5000	2	4	0.5000	A
2	1.000	2	2	1.000	B
2	1.000	2	8	1.000	B

---

Also we can suppress separation

```
(-> DS
  (api/group-by (juxt :V1 :V3))
  (api/ungroup {:separate? false
                :add-group-as-column true}))
;; => __unnamed [9 5]:
```

\_\_unnamed [9 5]:

:\$group-name	:V1	:V2	:V3	:V4
[1 1.0]	1	5	1.000	B
[1 0.5]	1	1	0.5000	A
[1 0.5]	1	7	0.5000	A
[2 1.5]	2	6	1.500	C
[1 1.5]	1	3	1.500	C
[1 1.5]	1	9	1.500	C
[2 0.5]	2	4	0.5000	A
[2 1.0]	2	2	1.000	B
[2 1.0]	2	8	1.000	B

## Other functions

To check if dataset is grouped or not just use `grouped?` function.

```
(api/grouped? DS)
```

nil

```
(api/grouped? (api/group-by DS :V1))
```

true

---

If you want to remove grouping annotation (to make all the functions work as with regular dataset) you can use `unmark-group` or `as-regular-dataset` (alias) functions.

It can be important when you want to remove some groups (rows) from grouped dataset using `drop-rows` or something like that.

```
(-> DS
  (api/group-by :V1)
  (api/as-regular-dataset)
  (api/grouped?))
```

nil

---

This is considered internal.

If you want to implement your own mapping function on grouped dataset you can call `process-group-data` and pass function operating on datasets. Result should be a dataset to have ungroupping working.

```
(-> DS
  (api/group-by :V1)
  (api/process-group-data #(str "Shape: " (vector (api/row-count %) (api/column-count %))))
  (api/as-regular-dataset))
```

\_\_unnamed [2 3]:

:name	:group-id	:data
1	0	Shape: [5 4]
2	1	Shape: [4 4]

## Columns

Column is a special `tech.ml.dataset` structure based on `tech.ml.datatype` library. For our purposes we can treat columns as typed and named sequence bound to particular dataset.

Type of the data is inferred from a sequence during column creation.

## Names

To select dataset columns or column names `columns-selector` is used. `columns-selector` can be one of the following:

- `:all` keyword - selects all columns
- column name - for single column
- sequence of column names - for collection of columns
- regex - to apply pattern on column names or datatype
- filter predicate - to filter column names or datatype

Column name can be anything.

`column-names` function returns names according to `columns-selector` and optional `meta-field`. `meta-field` is one of the following:

- `:name` (default) - to operate on column names
- `:datatype` - to operate on column types

- `:all` - if you want to process all metadata

---

To select all column names you can use `column-names` function.

```
(api/column-names DS)
```

```
(:V1 :V2 :V3 :V4)
```

or

```
(api/column-names DS :all)
```

```
(:V1 :V2 :V3 :V4)
```

In case you want to select column which has name `:all` (or is sequence or map), put it into a vector. Below code returns empty sequence since there is no such column in the dataset.

```
(api/column-names DS [:all])
```

```
()
```

---

Obviously selecting single name returns it's name if available

```
(api/column-names DS :V1)
```

```
(api/column-names DS "no such column")
```

```
(:V1)
```

```
()
```

---

Select sequence of column names.

```
(api/column-names DS [:V1 "V2" :V3 :V4 :V5])
```

```
(:V1 :V3 :V4)
```

---

Select names based on regex, columns ends with 1 or 4

```
(api/column-names DS #"^[14]$")
```

```
(:V1 :V4)
```

---

Select names based on regex operating on type of the column (to check what are the column types, call `(api/info DS :columns)`). Here we want to get integer columns only.

```
(api/column-names DS #"^:int.*" :datatype)
```

```
(:V1 :V2)
```

---

And finally we can use predicate to select names. Let's select double precision columns.

```
(api/column-names DS #(= :float64 %) :datatype)
```

```
(:V3)
```

If you want to select all columns but given, use `complement` function. Works only on a predicate.

```
(api/column-names DS (complement #{:V1}))  
(api/column-names DS (complement #(= :float64 %)) :datatype)
```

```
(:V2 :V3 :V4)  
(:V1 :V2 :V4)
```

---

You can select column names based on all column metadata at once by using `:all` metadata selector. Below we want to select column names ending with 1 which have long datatype.

```
(api/column-names DS (fn [meta]  
  (and (= :int64 (:datatype meta))  
        (clojure.string/ends-with? (:name meta) "1")))) :all)
```

```
(:V1)
```

## Select

`select-columns` creates dataset with columns selected by `columns-selector` as described above. Function works on regular and grouped dataset.

---

Select only float64 columns

```
(api/select-columns DS #(= :float64 %) :datatype)
```

\_\_unnamed [9 1]:

	:V3
0.5000	
1.000	
1.500	
0.5000	
1.000	
1.500	
0.5000	
1.000	
1.500	

---

Select all but :V1 columns

```
(api/select-columns DS (complement #{:V1}))
```

\_\_unnamed [9 3]:

	:V2	:V3	:V4
1		0.5000	A
2		1.000	B
3		1.500	C
4		0.5000	A
5		1.000	B
6		1.500	C



:V2	:V3	:V4
7	0.5000	A
8	1.000	B
9	1.500	C

If we have grouped data set, column selection is applied to every group separately.

```
(-> DS
  (api/group-by :V1)
  (api/select-columns [ :V2 :V3 ])
  (api/groups->map))
```

{1 Group: 1 [5 2]:

:V2	:V3
1	0.5000
3	1.500
5	1.000
7	0.5000
9	1.500

, 2 Group: 2 [4 2]:

:V2	:V3
2	1.000
4	0.5000
6	1.500
8	1.000

}

## Drop

`drop-columns` creates dataset with removed columns.

Drop float64 columns

```
(api/drop-columns DS #(= :float64 %) :datatype)
```

\_\_unnamed [9 3]:

:V1	:V2	:V4
1	1	A
2	2	B
1	3	C
2	4	A
1	5	B
2	6	C

:V1	:V2	:V4
1	7	A
2	8	B
1	9	C

---

Drop all columns but :V1 and :V2

```
(api/drop-columns DS (complement #{:V1 :V2}))
```

\_\_unnamed [9 2]:

:V1	:V2
1	1
2	2
1	3
2	4
1	5
2	6
1	7
2	8
1	9

---

If we have grouped data set, column selection is applied to every group separately. Selected columns are dropped.

```
(-> DS
  (api/group-by :V1)
  (api/drop-columns [ :V2 :V3])
  (api/groups->map))
```

{1 Group: 1 [5 2]:

:V1	:V4
1	A
1	C
1	B
1	A
1	C

, 2 Group: 2 [4 2]:

:V1	:V4
2	B
2	A
2	C
2	B

```
}
```

## Rename

If you want to rename columns use `rename-columns` and pass map where keys are old names, values new ones.

```
(api/rename-columns DS {:V1 "v1"  
                        :V2 "v2"  
                        :V3 [1 2 3]  
                        :V4 (Object.)})
```

\_\_unnamed [9 4]:

v1	v2	[1 2 3]	java.lang.Object@d2c3416
1	1	0.5000	A
2	2	1.000	B
1	3	1.500	C
2	4	0.5000	A
1	5	1.000	B
2	6	1.500	C
1	7	0.5000	A
2	8	1.000	B
1	9	1.500	C

Function works on grouped dataset

```
(-> DS  
  (api/group-by :V1)  
  (api/rename-columns {:V1 "v1"  
                      :V2 "v2"  
                      :V3 [1 2 3]  
                      :V4 (Object.)})  
  
  (api/groups->map))
```

{1 Group: 1 [5 4]:

v1	v2	[1 2 3]	java.lang.Object@4165c5f9
1	1	0.5000	A
1	3	1.500	C
1	5	1.000	B
1	7	0.5000	A
1	9	1.500	C

, 2 Group: 2 [4 4]:

v1	v2	[1 2 3]	java.lang.Object@4165c5f9
2	2	1.000	B
2	4	0.5000	A
2	6	1.500	C

v1	v2	[1 2 3]	java.lang.Object@4165c5f9
2	8	1.000	B

}

## Add or update

To add (or update existing) column call `add-or-update-column` function. Function accepts:

- **ds** - a dataset
- **column-name** - if it's existing column name, column will be replaced
- **column** - can be column (from other dataset), sequence, single value or function. Too big columns are always trimmed. Too small are cycled or extended with missing values (according to **size-strategy** argument)
- **size-strategy** (optional) - when new column is shorter than dataset row count, following strategies are applied:
  - **:cycle** (default) - repeat data
  - **:na** - append missing values

Function works on grouped dataset.

Add single value as column

```
(api/add-or-update-column DS :V5 "X")
```

\_\_unnamed [9 5]:

:V1	:V2	:V3	:V4	:V5
1	1	0.5000	A	X
2	2	1.000	B	X
1	3	1.500	C	X
2	4	0.5000	A	X
1	5	1.000	B	X
2	6	1.500	C	X
1	7	0.5000	A	X
2	8	1.000	B	X
1	9	1.500	C	X

Replace one column (column is trimmed)

```
(api/add-or-update-column DS :V1 (repeatedly rand))
```

\_\_unnamed [9 4]:

:V1	:V2	:V3	:V4
0.4241	1	0.5000	A
0.8455	2	1.000	B
0.4797	3	1.500	C
0.8634	4	0.5000	A
0.2812	5	1.000	B
0.5213	6	1.500	C

---

:V1	:V2	:V3	:V4
0.6539	7	0.5000	A
0.2455	8	1.000	B
0.5489	9	1.500	C

---

Copy column

```
(api/add-or-update-column DS :V5 (DS :V1))
```

\_\_unnamed [9 5]:

---

:V1	:V2	:V3	:V4	:V5
1	1	0.5000	A	1
2	2	1.000	B	2
1	3	1.500	C	1
2	4	0.5000	A	2
1	5	1.000	B	1
2	6	1.500	C	2
1	7	0.5000	A	1
2	8	1.000	B	2
1	9	1.500	C	1

---

When function is used, argument is whole dataset and the result should be column, sequence or single value

```
(api/add-or-update-column DS :row-count api/row-count)
```

\_\_unnamed [9 5]:

---

:V1	:V2	:V3	:V4	:row-count
1	1	0.5000	A	9
2	2	1.000	B	9
1	3	1.500	C	9
2	4	0.5000	A	9
1	5	1.000	B	9
2	6	1.500	C	9
1	7	0.5000	A	9
2	8	1.000	B	9
1	9	1.500	C	9

---

Above example run on grouped dataset, applies function on each group separately.

```
(-> DS
  (api/group-by :V1)
  (api/add-or-update-column :row-count api/row-count)
  (api/ungroup))
```

\_\_unnamed [9 5]:

:V1	:V2	:V3	:V4	:row-count
1	1	0.5000	A	5
1	3	1.500	C	5
1	5	1.000	B	5
1	7	0.5000	A	5
1	9	1.500	C	5
2	2	1.000	B	4
2	4	0.5000	A	4
2	6	1.500	C	4
2	8	1.000	B	4

When column which is added is longer than row count in dataset, column is trimmed. When column is shorter, it's cycled or missing values are appended.

```
(api/add-or-update-column DS :V5 [:r :b])
```

\_\_unnamed [9 5]:

:V1	:V2	:V3	:V4	:V5
1	1	0.5000	A	:r
2	2	1.000	B	:b
1	3	1.500	C	:r
2	4	0.5000	A	:b
1	5	1.000	B	:r
2	6	1.500	C	:b
1	7	0.5000	A	:r
2	8	1.000	B	:b
1	9	1.500	C	:r

```
(api/add-or-update-column DS :V5 [:r :b] :na)
```

\_\_unnamed [9 5]:

:V1	:V2	:V3	:V4	:V5
1	1	0.5000	A	:r
2	2	1.000	B	:b
1	3	1.500	C	
2	4	0.5000	A	
1	5	1.000	B	
2	6	1.500	C	
1	7	0.5000	A	
2	8	1.000	B	
1	9	1.500	C	

The same applies for grouped dataset

```
(-> DS
  (api/group-by :V3)
  (api/add-or-update-column :V5 [:r :b] :na))
```

```
(api/ungroup))
```

\_\_unnamed [9 5]:

:V1	:V2	:V3	:V4	:V5
2	2	1.000	B	:r
1	5	1.000	B	:b
2	8	1.000	B	
1	1	0.5000	A	:r
2	4	0.5000	A	:b
1	7	0.5000	A	
1	3	1.500	C	:r
2	6	1.500	C	:b
1	9	1.500	C	

Let's use other column to fill groups

```
(-> DS
  (api/group-by :V3)
  (api/add-or-update-column :V5 (DS :V2))
  (api/ungroup))
```

\_\_unnamed [9 5]:

:V1	:V2	:V3	:V4	:V5
2	2	1.000	B	1
1	5	1.000	B	2
2	8	1.000	B	3
1	1	0.5000	A	1
2	4	0.5000	A	2
1	7	0.5000	A	3
1	3	1.500	C	1
2	6	1.500	C	2
1	9	1.500	C	3

In case you want to add or update several columns you can call `add-or-update-columns` and provide map where keys are column names, vals are columns.

```
(api/add-or-update-columns DS { :V1 #(map inc (% :V1))
                                :V5 #(map (comp keyword str) (% :V4))
                                :V6 11})
```

\_\_unnamed [9 6]:

:V1	:V2	:V3	:V4	:V5	:V6
2	1	0.5000	A	:A	11
3	2	1.000	B	:B	11
2	3	1.500	C	:C	11
3	4	0.5000	A	:A	11
2	5	1.000	B	:B	11

:V1	:V2	:V3	:V4	:V5	:V6
3	6	1.500	C	:C	11
2	7	0.5000	A	:A	11
3	8	1.000	B	:B	11
2	9	1.500	C	:C	11

## Map

The other way of creating or updating column is to map columns as regular `map` function. The arity of mapping function should be the same as number of selected columns.

Arguments:

- `ds` - dataset
- `column-name` - target column name
- `map-fn` - mapping function
- `columns-selector` - columns selected
- `meta-field` (optional) - column selector option

Let's add numerical columns together

```
(api/map-columns DS :sum-of-numbers (fn [& rows]
                                       (reduce + rows)) #{:int64 :float64} :datatype)
```

\_\_unnamed [9 5]:

:V1	:V2	:V3	:V4	:sum-of-numbers
1	1	0.5000	A	2.500
2	2	1.000	B	5.000
1	3	1.500	C	5.500
2	4	0.5000	A	6.500
1	5	1.000	B	7.000
2	6	1.500	C	9.500
1	7	0.5000	A	8.500
2	8	1.000	B	11.00
1	9	1.500	C	11.50

The same works on grouped dataset

```
(-> DS
  (api/group-by :V4)
  (api/map-columns :sum-of-numbers (fn [& rows]
                                       (reduce + rows)) #{:int64 :float64} :datatype)
  (api/ungroup))
```

\_\_unnamed [9 5]:

:V1	:V2	:V3	:V4	:sum-of-numbers
1	1	0.5000	A	2.500
2	4	0.5000	A	6.500
1	7	0.5000	A	8.500
2	2	1.000	B	5.000



:V1	:V2	:V3	:V4	:sum-of-numbers
1	5	1.000	B	7.000
2	8	1.000	B	11.00
1	3	1.500	C	5.500
2	6	1.500	C	9.500
1	9	1.500	C	11.50

## Reorder

To reorder columns use columns selectors to choose what columns go first. The unselected columns are appended to the end.

```
(api/reorder-columns DS :V4 [:V3 :V2] :V1)
```

\_\_unnamed [9 4]:

:V4	:V2	:V3	:V1
A	1	0.5000	1
B	2	1.000	2
C	3	1.500	1
A	4	0.5000	2
B	5	1.000	1
C	6	1.500	2
A	7	0.5000	1
B	8	1.000	2
C	9	1.500	1

This function doesn't let you select meta field, so you have to call `column-names` in such case. Below we want to add integer columns at the end.

```
(api/reorder-columns DS (api/column-names DS (complement #{:int64}) :datatype))
```

\_\_unnamed [9 4]:

:V3	:V4	:V1	:V2
0.5000	A	1	1
1.000	B	2	2
1.500	C	1	3
0.5000	A	2	4
1.000	B	1	5
1.500	C	2	6
0.5000	A	1	7
1.000	B	2	8
1.500	C	1	9

## Type conversion

To convert column into given datatype can be done using `convert-column-type` function. Not all the types can be converted automatically also some types require slow parsing (every conversion from string). In case where conversion is not possible you can pass conversion function.

Arguments:

- `ds` - dataset
- Two options:
  - `coltype-map` in case when you want to convert several columns, keys are column names, vals are new types
  - `colname` and `new-type` - column name and new datatype

`new-type` can be:

- a type like `:int64` or `:string`
- or pair of datetime and conversion function

After conversion additional information is given on problematic values.

The other conversion is casting column into java array (`->array`) of the type column or provided as argument. Grouped dataset returns sequence of arrays.

---

Basic conversion

```
(-> DS
  (api/convert-column-type :V1 :float64)
  (api/info :columns))
```

\_\_unnamed :column info [4 6]:

---

:name	:size	:datatype	:unparsed-indexes	:unparsed-data	:categorical?
:V1	9	:float64	{}	[]	
:V2	9	:int64			
:V3	9	:float64			
:V4	9	:string			true

---

Using custom converter. Let's treat `:V4` as hexadecimal values. See that this way we can map column to any value.

```
(-> DS
  (api/convert-column-type :V4 [[:int16 #(Integer/parseInt % 16)]]))
```

\_\_unnamed [9 4]:

---

	:V1	:V2	:V3	:V4
1	1	1	0.5000	10
2	2	2	1.000	11
1	3	3	1.500	12
2	4	4	0.5000	10
1	5	5	1.000	11
2	6	6	1.500	12
1	7	7	0.5000	10
2	8	8	1.000	11
1	9	9	1.500	12

---

You can process several columns at once

```
(-> DS
  (api/convert-column-type {:V1 :float64
                           :V2 :object
                           :V3 [[:boolean #(< % 1.0)]
                                :V4 :object]))
  (api/info :columns))
```

\_\_unnamed :column info [4 5]:

:name	:size	:datatype	:unparsed-indexes	:unparsed-data
:V1	9	:float64	{}	[]
:V2	9	:object	{}	[]
:V3	9	:boolean	{}	[]
:V4	9	:object		

Function works on the grouped dataset

```
(-> DS
  (api/group-by :V1)
  (api/convert-column-type :V1 :float32)
  (api/ungroup)
  (api/info :columns))
```

\_\_unnamed :column info [4 6]:

:name	:size	:datatype	:unparsed-indexes	:unparsed-data	:categorical?
:V1	9	:float32	{}	[]	
:V2	9	:int64			
:V3	9	:float64			
:V4	9	:string			true

Double array conversion.

```
(api/->array DS :V1)
```

```
#object["[J" 0x7627907 "[J@7627907"]
```

Function also works on grouped dataset

```
(-> DS
  (api/group-by :V3)
  (api/->array :V2))
```

```
(#object["[J" 0x64359c08 "[J@64359c08"] #object["[J" 0x719f6ea5 "[J@719f6ea5"] #object["[J" 0x4d031458 "[J@4d031458"]
```

You can also cast the type to the other one (if casting is possible):

```
(api/->array DS :V4 :string)
(api/->array DS :V1 :float32)
```

```
#object["[Ljava.lang.String;" 0x4050cf1 "[Ljava.lang.String;@4050cf1"]
#object["[F" 0x1f01a0a3 "[F@1f01a0a3"]
```

## Rows

Rows can be selected or dropped using various selectors:

- row id(s) - row index as number or sequence of numbers (first row has index 0, second 1 and so on)
- sequence of true/false values
- filter by predicate (argument is row as a map)

When predicate is used you may want to limit columns passed to the function (`limit-columns` option).

Additionally you may want to precalculate some values which will be visible for predicate as additional columns. It's done internally by calling `add-or-update-columns` on a dataset. `:pre` is used as a column definitions.

## Select

Select fourth row

```
(api/select-rows DS 4)
```

\_\_unnamed [1 4]:

:V1	:V2	:V3	:V4
1	5	1.000	B

Select 3 rows

```
(api/select-rows DS [1 4 5])
```

\_\_unnamed [3 4]:

:V1	:V2	:V3	:V4
2	2	1.000	B
1	5	1.000	B
2	6	1.500	C

Select rows using sequence of true/false values

```
(api/select-rows DS [true nil nil true])
```

\_\_unnamed [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	4	0.5000	A

Select rows using predicate

```
(api/select-rows DS (comp #(< % 1) :V3))
```

\_\_unnamed [3 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	4	0.5000	A
1	7	0.5000	A

---

The same works on grouped dataset, let's select first row from every group.

```
(-> DS  
  (api/group-by :V1)  
  (api/select-rows 0)  
  (api/ungroup))
```

\_\_unnamed [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	2	1.000	B

---

If you want to select :V2 values which are lower than or equal mean in grouped dataset you have to precalculate it using :pre.

```
(-> DS  
  (api/group-by :V4)  
  (api/select-rows (fn [row] (<= (:V2 row) (:mean row)))  
    {:pre {:mean #(tech.v2.datatype.functional/mean (% :V2))}})  
  (api/ungroup))
```

\_\_unnamed [6 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	4	0.5000	A
2	2	1.000	B
1	5	1.000	B
1	3	1.500	C
2	6	1.500	C

## Drop

drop-rows removes rows, and accepts exactly the same parameters as select-rows

---

Drop values lower than or equal :V2 column mean in grouped dataset.

```
(-> DS
  (api/group-by :V4)
  (api/drop-rows (fn [row] (<= (:V2 row) (:mean row))))
  {:pre {:mean #(tech.v2.datatype.functional/mean (% :V2))}})
  (api/ungroup))
```

\_\_unnamed [3 4]:

---

:V1	:V2	:V3	:V4
1	7	0.5000	A
2	8	1.000	B
1	9	1.500	C

---

## Other

There are several function to select first, last, random rows, or display head, tail of the dataset. All functions work on grouped dataset.

---

First row

```
(api/first DS)
```

\_\_unnamed [1 4]:

---

:V1	:V2	:V3	:V4
1	1	0.5000	A

---



---

Last row

```
(api/last DS)
```

\_\_unnamed [1 4]:

---

:V1	:V2	:V3	:V4
1	9	1.500	C

---



---

Random row (single)

```
(api/rand-nth DS)
```

\_\_unnamed [1 4]:

---

:V1	:V2	:V3	:V4
2	6	1.500	C

---



---

Random **n** (default: row count) rows with repetition.

```
(api/random DS)
```

```
__unnamed [9 4]:
```

:V1	:V2	:V3	:V4
2	6	1.500	C
1	7	0.5000	A
1	9	1.500	C
2	4	0.5000	A
1	9	1.500	C
1	5	1.000	B
1	9	1.500	C
2	4	0.5000	A
2	6	1.500	C

---

Five random rows with repetition

```
(api/random DS 5)
```

```
__unnamed [5 4]:
```

:V1	:V2	:V3	:V4
1	5	1.000	B
2	6	1.500	C
2	8	1.000	B
1	1	0.5000	A
1	7	0.5000	A

---

Five random, non-repeating rows

```
(api/random DS 5 {:repeat? false})
```

```
__unnamed [5 4]:
```

:V1	:V2	:V3	:V4
1	9	1.500	C
1	7	0.5000	A
1	3	1.500	C
2	8	1.000	B
2	6	1.500	C

---

Shuffle dataset

```
(api/shuffle DS)
```

```
__unnamed [9 4]:
```

:V1	:V2	:V3	:V4
2	8	1.000	B
1	3	1.500	C
2	2	1.000	B
1	7	0.5000	A
1	1	0.5000	A
2	6	1.500	C
1	5	1.000	B
2	4	0.5000	A
1	9	1.500	C

---

First **n** rows (default 5)

(api/head DS)

\_\_unnamed [5 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	2	1.000	B
1	3	1.500	C
2	4	0.5000	A
1	5	1.000	B

---

Last **n** rows (default 5)

(api/tail DS)

\_\_unnamed [5 4]:

:V1	:V2	:V3	:V4
1	5	1.000	B
2	6	1.500	C
1	7	0.5000	A
2	8	1.000	B
1	9	1.500	C

---

Select 5 random rows from each group

```
(-> DS
  (api/group-by :V4)
  (api/random 5)
  (api/ungroup))
```

\_\_unnamed [15 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A



:V1	:V2	:V3	:V4
1	1	0.5000	A
1	7	0.5000	A
2	4	0.5000	A
1	1	0.5000	A
2	2	1.000	B
2	8	1.000	B
1	5	1.000	B
2	2	1.000	B
1	5	1.000	B
1	3	1.500	C
2	6	1.500	C
2	6	1.500	C
1	3	1.500	C
2	6	1.500	C

## Aggregate

Aggregating is a function which produces single row out of dataset.

Aggregator is a function or sequence or map of functions which accept dataset as an argument and result single value, sequence of values or map.

Where map is given as an input or result, keys are treated as column names.

Grouped dataset is ungrouped after aggregation. This can be turned off by setting `:ungroup` to false. In case you want to pass additional ungrouping parameters add them to the options.

By default resulting column names are prefixed with `summary` prefix (set it with `:default-column-name-prefix` option).

---

Let's calculate mean of some columns

```
(api/aggregate DS #(reduce + (% :V2)))
```

\_\_unnamed [1 1]:

:summary
45

---

Let's give resulting column a name.

```
(api/aggregate DS {:sum-of-V2 #(reduce + (% :V2))})
```

\_\_unnamed [1 1]:

:sum-of-V2
45

---

Sequential result is spread into separate columns

```
(api/aggregate DS #(take 5(% :V2)))
```

\_\_unnamed [1 5]:

:summary-0	:summary-1	:summary-2	:summary-3	:summary-4
1	2	3	4	5

You can combine all variants and rename default prefix

```
(api/aggregate DS [(take 3 (% :V2))
  (fn [ds] {:sum-v1 (reduce + (ds :V1))
            :prod-v3 (reduce * (ds :V3))})] {:default-column-name-prefix "V2-value"}))
```

\_\_unnamed [1 5]:

:V2-value-0-0	:V2-value-0-1	:V2-value-0-2	:sum-v1	:prod-v3
1	2	3	13	0.4219

Processing grouped dataset

```
(-> DS
  (api/group-by [:V4])
  (api/aggregate [(take 3 (% :V2))
    (fn [ds] {:sum-v1 (reduce + (ds :V1))
              :prod-v3 (reduce * (ds :V3))})] {:default-column-name-prefix "V2-value"}))
```

\_\_unnamed [3 6]:

:V4	:V2-value-0-0	:V2-value-0-1	:V2-value-0-2	:sum-v1	:prod-v3
B	2	5	8	5	1.000
C	3	6	9	4	3.375
A	1	4	7	4	0.1250

Result of aggregating is automatically ungrouped, you can skip this step by setting :ungroup option to false.

```
(-> DS
  (api/group-by [:V3])
  (api/aggregate [(take 3 (% :V2))
    (fn [ds] {:sum-v1 (reduce + (ds :V1))
              :prod-v3 (reduce * (ds :V3))})] {:default-column-name-prefix "V2-value"
        :ungroup? false}))
```

\_\_unnamed [3 3]:

:name	:group-id	:data
{:V3 1.0}	0	__unnamed [1 5]:
{:V3 0.5}	1	__unnamed [1 5]:

---

:name	:group-id	:data
{:V3 1.5}	2	__unnamed [1 5]:

---

## Order

Ordering can be done by column(s) or any function operating on row. Possible order can be:

- `:asc` for ascending order (default)
- `:desc` for descending order
- custom comparator

`:limit-columns` limits row map provided to ordering functions.

---

Order by single column, ascending

```
(api/order-by DS :V1)
```

\_\_unnamed [9 4]:

---

:V1	:V2	:V3	:V4
1	1	0.5000	A
1	3	1.500	C
1	5	1.000	B
1	7	0.5000	A
1	9	1.500	C
2	6	1.500	C
2	4	0.5000	A
2	8	1.000	B
2	2	1.000	B

---

Descending order

```
(api/order-by DS :V1 :desc)
```

\_\_unnamed [9 4]:

---

:V1	:V2	:V3	:V4
2	2	1.000	B
2	4	0.5000	A
2	6	1.500	C
2	8	1.000	B
1	5	1.000	B
1	3	1.500	C
1	7	0.5000	A
1	1	0.5000	A
1	9	1.500	C

---

Order by two columns

```
(api/order-by DS [ :V1 :V2])
```

\_\_unnamed [9 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
1	3	1.500	C
1	5	1.000	B
1	7	0.5000	A
1	9	1.500	C
2	2	1.000	B
2	4	0.5000	A
2	6	1.500	C
2	8	1.000	B

Use different orders for columns

```
(api/order-by DS [ :V1 :V2] [ :asc :desc])
```

\_\_unnamed [9 4]:

:V1	:V2	:V3	:V4
1	9	1.500	C
1	7	0.5000	A
1	5	1.000	B
1	3	1.500	C
1	1	0.5000	A
2	8	1.000	B
2	6	1.500	C
2	4	0.5000	A
2	2	1.000	B

```
(api/order-by DS [ :V1 :V2] [ :desc :desc])
```

\_\_unnamed [9 4]:

:V1	:V2	:V3	:V4
2	8	1.000	B
2	6	1.500	C
2	4	0.5000	A
2	2	1.000	B
1	9	1.500	C
1	7	0.5000	A
1	5	1.000	B
1	3	1.500	C
1	1	0.5000	A

```
(api/order-by DS [ :V1 :V3] [ :desc :asc])
```

\_\_unnamed [9 4]:

:V1	:V2	:V3	:V4
2	4	0.5000	A
2	2	1.000	B
2	8	1.000	B
2	6	1.500	C
1	1	0.5000	A
1	7	0.5000	A
1	5	1.000	B
1	3	1.500	C
1	9	1.500	C

---

Custom function can be used to provide ordering key. Here order by :V4 descending, then by product of other columns ascending.

```
(api/order-by DS [ :V4 (fn [row] (* (:V1 row)
                                   (:V2 row)
                                   (:V3 row)))] [ :desc :asc] { :limit-columns [ :V1 :V2 :V3 ]})
```

\_\_unnamed [9 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
1	7	0.5000	A
2	4	0.5000	A
2	2	1.000	B
1	3	1.500	C
1	5	1.000	B
1	9	1.500	C
2	8	1.000	B
2	6	1.500	C

---

Custom comparator also can be used in case objects are not comparable by default. Let's define artificial one: if euclidean distance is lower than 2, compare along z else along x and y. We use first three columns for that.

```
(defn dist
  [v1 v2]
  (->> v2
    (map - v1)
    (map #(* % %))
    (reduce +)
    (Math/sqrt)))
```

#'user/dist

```
(api/order-by DS [ :V1 :V2 :V3 ] (fn [[x1 y1 z1 :as v1] [x2 y2 z2 :as v2]]
  (let [d (dist v1 v2)]
    (if (< d 2.0)
      (compare z1 z2)
```

```
(compare [x1 y1] [x2 y2]))))
```

\_\_unnamed [9 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
1	5	1.000	B
1	7	0.5000	A
1	9	1.500	C
2	2	1.000	B
2	4	0.5000	A
1	3	1.500	C
2	6	1.500	C
2	8	1.000	B

## Unique

Remove rows which contains the same data. By default **unique-by** removes duplicates from whole dataset. You can also pass list of columns or functions (similar as in **group-by**) to remove duplicates limited by them. Default strategy is to keep the first row. More strategies below.

**unique-by** works on groups

---

Remove duplicates from whole dataset

```
(api/unique-by DS)
```

\_\_unnamed [9 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	2	1.000	B
1	3	1.500	C
2	4	0.5000	A
1	5	1.000	B
2	6	1.500	C
1	7	0.5000	A
2	8	1.000	B
1	9	1.500	C

---

Remove duplicates from each group selected by column.

```
(api/unique-by DS :V1)
```

\_\_unnamed [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	2	1.000	B

---

Pair of columns

```
(api/unique-by DS [ :V1 :V3])
```

\_\_unnamed [6 4]:

---

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	2	1.000	B
1	3	1.500	C
2	4	0.5000	A
1	5	1.000	B
2	6	1.500	C

---

---

Also function can be used, split dataset by modulo 3 on columns :V2

```
(api/unique-by DS (fn [m] (mod (:V2 m) 3)))
```

\_\_unnamed [3 4]:

---

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	2	1.000	B
1	3	1.500	C

---

---

The same can be achieved with group-by

```
(-> DS  
  (api/group-by (fn [m] (mod (:V2 m) 3)))  
  (api/first)  
  (api/ungroup))
```

\_\_unnamed [3 4]:

---

:V1	:V2	:V3	:V4
1	3	1.500	C
1	1	0.5000	A
2	2	1.000	B

---

---

Grouped dataset

```
(-> DS  
  (api/group-by :V4)  
  (api/unique-by :V1)  
  (api/ungroup))
```

\_\_unnamed [6 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	4	0.5000	A
2	2	1.000	B
1	5	1.000	B
1	3	1.500	C
2	6	1.500	C

## Strategies

There are 4 strategies defined:

- `:first` - select first row (default)
- `:last` - select last row
- `:random` - select random row
- any function - apply function to a columns which are subject of uniqueness

Last

```
(api/unique-by DS :V1 {:strategy :last})
```

\_\_unnamed [2 4]:

:V1	:V2	:V3	:V4
2	8	1.000	B
1	9	1.500	C

Random

```
(api/unique-by DS :V1 {:strategy :random})
```

\_\_unnamed [2 4]:

:V1	:V2	:V3	:V4
2	6	1.500	C
1	7	0.5000	A

Pack columns into vector

```
(api/unique-by DS :V4 {:strategy vec})
```

\_\_unnamed [3 3]:

:V1	:V2	:V3
[2 1 2]	[2 5 8]	[1.0 1.0 1.0]
[1 2 1]	[3 6 9]	[1.5 1.5 1.5]
[1 2 1]	[1 4 7]	[0.5 0.5 0.5]



---

Sum columns

```
(api/unique-by DS :V4 {:strategy (partial reduce +)}))
```

\_\_unnamed [3 3]:

:V1	:V2	:V3
5	15	3.000
4	18	4.500
4	12	1.500

---

Group by function and apply functions

```
(api/unique-by DS (fn [m] (mod (:V2 m) 3)) {:strategy vec}))
```

\_\_unnamed [3 4]:

:V1	:V2	:V3	:V4
[1 2 1]	[3 6 9]	[1.5 1.5 1.5]	["C" "C" "C"]
[1 2 1]	[1 4 7]	[0.5 0.5 0.5]	["A" "A" "A"]
[2 1 2]	[2 5 8]	[1.0 1.0 1.0]	["B" "B" "B"]

---

Grouped dataset

```
(-> DS  
  (api/group-by :V1)  
  (api/unique-by (fn [m] (mod (:V2 m) 3)) {:strategy vec}))  
  (api/ungroup {:add-group-as-column :from-V1}))
```

\_\_unnamed [6 5]:

:from-V1	:V1	:V2	:V3	:V4
1	[1 1]	[3 9]	[1.5 1.5]	["C" "C"]
1	[1 1]	[1 7]	[0.5 0.5]	["A" "A"]
1	[1]	[5]	[1.0]	["B"]
2	[2]	[6]	[1.5]	["C"]
2	[2]	[4]	[0.5]	["A"]
2	[2 2]	[2 8]	[1.0 1.0]	["B" "B"]

## Missing

When dataset contains missing values you can select or drop rows with missing values or replace them using some strategy.

**column-selector** can be used to limit considered columns

Let's define dataset which contains missing values

```
(def DSm (api/dataset { :V1 (take 9 (cycle [1 2 nil]))
                        :V2 (range 1 10)
                        :V3 (take 9 (cycle [0.5 1.0 nil 1.5]))
                        :V4 (take 9 (cycle ["A" "B" "C"]))}))
```

DSm

\_\_unnamed [9 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	2	1.000	B
	3		C
1	4	1.500	A
2	5	0.5000	B
	6	1.000	C
1	7		A
2	8	1.500	B
	9	0.5000	C

## Select

Select rows with missing values

```
(api/select-missing DSm)
```

\_\_unnamed [4 4]:

:V1	:V2	:V3	:V4
	3		C
	6	1.000	C
1	7		A
	9	0.5000	C

Select rows with missing values in :V1

```
(api/select-missing DSm :V1)
```

\_\_unnamed [3 4]:

:V1	:V2	:V3	:V4
	3		C
	6	1.000	C
	9	0.5000	C

The same with grouped dataset

```
(-> DSm
  (api/group-by :V4)
  (api/select-missing :V3))
```

```
(api/ungroup))
```

\_\_unnamed [2 4]:

:V1	:V2	:V3	:V4
1	7		A
	3		C

## Drop

Drop rows with missing values

```
(api/drop-missing DSm)
```

\_\_unnamed [5 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	2	1.000	B
1	4	1.500	A
2	5	0.5000	B
2	8	1.500	B

---

Drop rows with missing values in :V1

```
(api/drop-missing DSm :V1)
```

\_\_unnamed [6 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
2	2	1.000	B
1	4	1.500	A
2	5	0.5000	B
1	7		A
2	8	1.500	B

---

The same with grouped dataset

```
(-> DSm  
  (api/group-by :V4)  
  (api/drop-missing :V1)  
  (api/ungroup))
```

\_\_unnamed [6 4]:

:V1	:V2	:V3	:V4
1	1	0.5000	A
1	4	1.500	A

:V1	:V2	:V3	:V4
1	7		A
2	2	1.000	B
2	5	0.5000	B
2	8	1.500	B

## Replace

Missing values can be replaced using several strategies. `replace-missing` accepts:

- dataset
- column selector
- value
  - single value
  - sequence of values (cycled)
  - function, applied on column(s) with stripped missings
- strategy (optional)

Strategies are:

- `:value` - replace with given value (default)
- `:up` - copy values up
- `:down` - copy values down

Let's define special dataset here:

```
(def DSm2 (api/dataset { :a [nil nil nil 1.0 2 nil 4 nil 11 nil nil]
                        :b [2.0 2 2 nil nil 3 nil 3 4 5 5]}))
```

DSm2

\_\_unnamed [11 2]:

:a	:b
	2.000
	2.000
	2.000
1.000	
2.000	
	3.000
4.000	
	3.000
11.00	4.000
	5.000
	5.000

Replace missing with single value in whole dataset

```
(api/replace-missing DSm2 999)
```

\_\_unnamed [11 2]:

:a	:b
999.0	2.000

---

:a	:b
999.0	2.000
999.0	2.000
1.000	999.0
2.000	999.0
999.0	3.000
4.000	999.0
999.0	3.000
11.00	4.000
999.0	5.000
999.0	5.000

---



---

Replace missing with single value in :a column

```
(api/replace-missing DSm2 :a 999)
```

\_\_unnamed [11 2]:

---

:a	:b
999.0	2.000
999.0	2.000
999.0	2.000
1.000	
2.000	
999.0	3.000
4.000	
999.0	3.000
11.00	4.000
999.0	5.000
999.0	5.000

---



---

Replace missing with sequence in :a column

```
(api/replace-missing DSm2 :a [-999 -998 -997])
```

\_\_unnamed [11 2]:

---

:a	:b
-999.0	2.000
-998.0	2.000
-997.0	2.000
1.000	
2.000	
-999.0	3.000
4.000	
-998.0	3.000
11.00	4.000
-997.0	5.000
-999.0	5.000

---

---

Replace missing with a function (mean)

```
(api/replace-missing DSm2 :a tech.v2.datatype.functional/mean)
```

\_\_unnamed [11 2]:

:a	:b
4.500	2.000
4.500	2.000
4.500	2.000
1.000	
2.000	
4.500	3.000
4.000	
4.500	3.000
11.00	4.000
4.500	5.000
4.500	5.000

---

Using :down strategy, fills gaps with values from above. You can see that if missings are at the beginning, they are left missing.

```
(api/replace-missing DSm2 [:a :b] nil :down)
```

\_\_unnamed [11 2]:

:a	:b
	2.000
	2.000
	2.000
1.000	2.000
2.000	2.000
2.000	3.000
4.000	3.000
4.000	3.000
11.00	4.000
11.00	5.000
11.00	5.000

---

To fix above issue you can provide value

```
(api/replace-missing DSm2 [:a :b] 999 :down)
```

\_\_unnamed [11 2]:

:a	:b
999.0	2.000
999.0	2.000
999.0	2.000

:a	:b
1.000	2.000
2.000	2.000
2.000	3.000
4.000	3.000
4.000	3.000
11.00	4.000
11.00	5.000
11.00	5.000

---

The same applies for `:up` strategy which is opposite direction.

```
(api/replace-missing DSm2 [:a :b] 999 :up)
```

\_\_unnamed [11 2]:

:a	:b
1.000	2.000
1.000	2.000
1.000	2.000
1.000	3.000
2.000	3.000
4.000	3.000
4.000	3.000
11.00	3.000
11.00	4.000
999.0	5.000
999.0	5.000

---

We can use a function which is applied after applying `:up` or `:down`

```
(api/replace-missing DSm2 [:a :b] tech.v2.datatype.functional/mean :down)
```

\_\_unnamed [11 2]:

:a	:b
4.500	2.000
4.500	2.000
4.500	2.000
1.000	2.000
2.000	2.000
2.000	3.000
4.000	3.000
4.000	3.000
11.00	4.000
11.00	5.000
11.00	5.000

## Join/Separate Columns

Joining or separating columns are operations which can help to tidy messy dataset.

- `join-columns` joins content of the columns (as string concatenation or other structure) and stores it in new column
- `separate-column` splits content of the columns into set of new columns

### Join

`join-columns` accepts:

- dataset
- column selector (as in `select-columns`)
- options
  - `:separator` (default "-")
  - `:drop-columns?` - whether to drop source columns or not (default `true`)
  - `:result-type`
    - \* `:map` - packs data into map
    - \* `:seq` - packs data into sequence
    - \* `:string` - join strings with separator (default)
    - \* or custom function which gets row as a vector
  - `:missing-subst` - substitution for missing value

---

Default usage. Create `:joined` column out of other columns.

```
(api/join-columns DSm :joined [:V1 :V2 :V4])
```

\_\_unnamed [9 2]:

:V3	:joined
0.5000	1-1-A
1.000	2-2-B
	3-C
1.500	1-4-A
0.5000	2-5-B
1.000	6-C
	1-7-A
1.500	2-8-B
0.5000	9-C

---

Without dropping source columns.

```
(api/join-columns DSm :joined [:V1 :V2 :V4] {:drop-columns? false})
```

\_\_unnamed [9 5]:

:V1	:V2	:V3	:V4	:joined
1	1	0.5000	A	1-1-A
2	2	1.000	B	2-2-B
	3		C	3-C
1	4	1.500	A	1-4-A



:V1	:V2	:V3	:V4	:joined
2	5	0.5000	B	2-5-B
	6	1.000	C	6-C
1	7		A	1-7-A
2	8	1.500	B	2-8-B
	9	0.5000	C	9-C

Let's replace missing value with "NA" string.

```
(api/join-columns DSm :joined [:V1 :V2 :V4] {:missing-subst "NA"})
```

\_\_unnamed [9 2]:

:V3	:joined
0.5000	1-1-A
1.000	2-2-B
	NA-3-C
1.500	1-4-A
0.5000	2-5-B
1.000	NA-6-C
	1-7-A
1.500	2-8-B
0.5000	NA-9-C

We can use custom separator.

```
(api/join-columns DSm :joined [:V1 :V2 :V4] {:separator "/"
                                              :missing-subst "."})
```

\_\_unnamed [9 2]:

:V3	:joined
0.5000	1/1/A
1.000	2/2/B
	./3/C
1.500	1/4/A
0.5000	2/5/B
1.000	./6/C
	1/7/A
1.500	2/8/B
0.5000	./9/C

Or even sequence of separators.

```
(api/join-columns DSm :joined [:V1 :V2 :V4] {:separator ["-" "/"]
                                              :missing-subst "."})
```

\_\_unnamed [9 2]:

:V3	:joined
0.5000	1-1/A
1.000	2-2/B
	.-3/C
1.500	1-4/A
0.5000	2-5/B
1.000	.-6/C
	1-7/A
1.500	2-8/B
0.5000	.-9/C

The other types of results, map:

```
(api/join-columns DSm :joined [:V1 :V2 :V4] {:result-type :map})
```

\_\_unnamed [9 2]:

:V3	:joined
0.5000	{:V1 1, :V2 1, :V4 "A"}
1.000	{:V1 2, :V2 2, :V4 "B"}
	{:V1 nil, :V2 3, :V4 "C"}
1.500	{:V1 1, :V2 4, :V4 "A"}
0.5000	{:V1 2, :V2 5, :V4 "B"}
1.000	{:V1 nil, :V2 6, :V4 "C"}
	{:V1 1, :V2 7, :V4 "A"}
1.500	{:V1 2, :V2 8, :V4 "B"}
0.5000	{:V1 nil, :V2 9, :V4 "C"}

Sequence

```
(api/join-columns DSm :joined [:V1 :V2 :V4] {:result-type :seq})
```

\_\_unnamed [9 2]:

:V3	:joined
0.5000	(1 1 "A")
1.000	(2 2 "B")
	(nil 3 "C")
1.500	(1 4 "A")
0.5000	(2 5 "B")
1.000	(nil 6 "C")
	(1 7 "A")
1.500	(2 8 "B")
0.5000	(nil 9 "C")

Custom function, calculate hash

```
(api/join-columns DSm :joined [:V1 :V2 :V4] {:result-type hash}))
```

\_\_unnamed [9 2]:

:V3	:joined
0.5000	535226087
1.000	1128801549
	-1842240303
1.500	2022347171
0.5000	1884312041
1.000	-1555412370
	1640237355
1.500	-967279152
0.5000	1128367958

Grouped dataset

```
(-> DSm
  (api/group-by :V4)
  (api/join-columns :joined [:V1 :V2 :V4])
  (api/ungroup))
```

\_\_unnamed [9 2]:

:V3	:joined
0.5000	1-1-A
1.500	1-4-A
	1-7-A
1.000	2-2-B
0.5000	2-5-B
1.500	2-8-B
	3-C
1.000	6-C
0.5000	9-C

## Tidyr examples

source

```
(def df (api/dataset {:x ["a" "a" nil nil]
                      :y ["b" nil "b" nil]}))
```

#'user/df

df

\_\_unnamed [4 2]:

:x	:y
a	b

:x	:y
a	b

```
(api/join-columns df "z" [:x :y] {:drop-columns? false
                                   :missing-subst "NA"
                                   :separator "_"})
```

\_\_unnamed [4 3]:

:x	:y	z
a	b	a_b
a		a_NA
	b	NA_b
		NA_NA

```
(api/join-columns df "z" [:x :y] {:drop-columns? false
                                   :separator "_"})
```

\_\_unnamed [4 3]:

:x	:y	z
a	b	a_b
a		a
	b	b

## Separate

Column can be also separated into several other columns using string as separator, regex or custom function. Arguments:

- dataset
- source column
- target columns
- separator as:
  - string - it's converted to regular expression and passed to `clojure.string/split` function
  - regex
  - or custom function (default: identity)
- options
  - `:drop-columns?` - whether drop source column or not (default: `true`)
  - `:missing-subst` - values which should be treated as missing, can be set, sequence, value or function (default: `""`)

Custom function (as separator) should return sequence of values for given value.

Separate float into integer and factional values

```
(api/separate-column DS :V3 [:int-part :frac-part] (fn [~double v]
  [(int (quot v 1.0))
   (mod v 1.0)]))
```

\_\_unnamed [9 5]:

:V1	:V2	:int-part	:frac-part	:V4
1	1	0	0.5000	A
2	2	1	0.000	B
1	3	1	0.5000	C
2	4	0	0.5000	A
1	5	1	0.000	B
2	6	1	0.5000	C
1	7	0	0.5000	A
2	8	1	0.000	B
1	9	1	0.5000	C

Source column can be kept

```
(api/separate-column DS :V3 [:int-part :frac-part] (fn [~double v]
  [(int (quot v 1.0))
   (mod v 1.0)]} {:drop-column? false}))
```

\_\_unnamed [9 6]:

:V1	:V2	:V3	:int-part	:frac-part	:V4
1	1	0.5000	0	0.5000	A
2	2	1.000	1	0.000	B
1	3	1.500	1	0.5000	C
2	4	0.5000	0	0.5000	A
1	5	1.000	1	0.000	B
2	6	1.500	1	0.5000	C
1	7	0.5000	0	0.5000	A
2	8	1.000	1	0.000	B
1	9	1.500	1	0.5000	C

We can treat 0 or 0.0 as missing value

```
(api/separate-column DS :V3 [:int-part :frac-part] (fn [~double v]
  [(int (quot v 1.0))
   (mod v 1.0)]} {:missing-subst [0 0.0]}))
```

\_\_unnamed [9 5]:

:V1	:V2	:int-part	:frac-part	:V4
1	1		0.5000	A
2	2	1		B
1	3	1	0.5000	C
2	4		0.5000	A
1	5	1		B

:V1	:V2	:int-part	:frac-part	:V4
2	6	1	0.5000	C
1	7		0.5000	A
2	8	1		B
1	9	1	0.5000	C

Works on grouped dataset

```
(-> DS
  (api/group-by :V4)
  (api/separate-column :V3 [:int-part :frac-part] (fn [^double v]
                                                    [(int (quot v 1.0))
                                                     (mod v 1.0)]))
  (api/ungroup))
```

\_\_unnamed [9 5]:

:V1	:V2	:int-part	:frac-part	:V4
1	1	0	0.5000	A
2	4	0	0.5000	A
1	7	0	0.5000	A
2	2	1	0.000	B
1	5	1	0.000	B
2	8	1	0.000	B
1	3	1	0.5000	C
2	6	1	0.5000	C
1	9	1	0.5000	C

Join and separate together.

```
(-> DSm
  (api/join-columns :joined [:V1 :V2 :V4] {:result-type :map})
  (api/separate-column :joined [:v1 :v2 :v4] (juxt :V1 :V2 :V4)))
```

\_\_unnamed [9 4]:

:V3	:v1	:v2	:v4
0.5000	1	1	A
1.000	2	2	B
		3	C
1.500	1	4	A
0.5000	2	5	B
1.000		6	C
	1	7	A
1.500	2	8	B
0.5000		9	C

```
(-> DSm
  (api/join-columns :joined [:V1 :V2 :V4] {:result-type :seq})
  (api/separate-column :joined [:v1 :v2 :v4] identity))
```

\_\_unnamed [9 4]:

:V3	:v1	:v2	:v4
0.5000	1	1	A
1.000	2	2	B
		3	C
1.500	1	4	A
0.5000	2	5	B
1.000		6	C
	1	7	A
1.500	2	8	B
0.5000		9	C

## Tidyr examples

separate source extract source

```
(def df-separate (api/dataset {:x [nil "a.b" "a.d" "b.c"]}))
(def df-separate2 (api/dataset {:x ["a" "a b" nil "a b c"]}))
(def df-separate3 (api/dataset {:x ["a?b" nil "a.b" "b:c"]}))
(def df-extract (api/dataset {:x [nil "a-b" "a-d" "b-c" "d-e"]}))
```

```
#'user/df-separate
#'user/df-separate2
#'user/df-separate3
#'user/df-extract
```

df-separate

\_\_unnamed [4 1]:

```
____
:x
____
a.b
a.d
b.c
____
```

df-separate2

\_\_unnamed [4 1]:

```
____
:x
____
a
a b

a b c
____
```

```
df-separate3
```

```
__unnamed [4 1]:
```

```
_____  
:x  
_____  
a?b  
  
a.b  
b:c  
_____
```

```
df-extract
```

```
__unnamed [5 1]:
```

```
_____  
:x  
_____  
  
a-b  
a-d  
b-c  
d-e  
_____
```

---

```
(api/separate-column df-separate :x [:A :B] "\\\\.")
```

```
__unnamed [4 2]:
```

```
_____  
:A  :B  
_____  
  
a    b  
a    d  
b    c  
_____
```

---

You can drop columns after separation by setting `nil` as a name. We need second value here.

```
(api/separate-column df-separate :x [nil :B] "\\\\.")
```

```
__unnamed [4 1]:
```

```
_____  
:B  
_____  
  
b  
d  
c  
_____
```

---

Extra data is dropped



```
(api/separate-column df-separate2 :x ["a" "b"] " ")
```

\_\_unnamed [4 2]:

_____	
a	b
_____	
a	
a	b
_____	
a	b

Split with regular expression

```
(api/separate-column df-separate3 :x ["a" "b"] "[?\\\\.:]")
```

\_\_unnamed [4 2]:

_____	
a	b
_____	
a	b
a	b
b	c
_____	

Or just regular expression to extract values

```
(api/separate-column df-separate3 :x ["a" "b"] #"(.).(.)")
```

\_\_unnamed [4 2]:

_____	
a	b
_____	
a	b
a	b
b	c
_____	

Extract first value only

```
(api/separate-column df-extract :x ["A"] "-")
```

\_\_unnamed [5 1]:

_____
A
_____
a
a
b
d
_____

---

Split with regex

```
(api/separate-column df-extract :x ["A" "B"] #"(\p{Alnum})-(\p{Alnum})")
```

\_\_unnamed [5 2]:

A	B
a	b
a	d
b	c
d	e

---

Only a,b,c,d strings

```
(api/separate-column df-extract :x ["A" "B"] #"([a-d]+)-([a-d]+)")
```

\_\_unnamed [5 2]:

A	B
a	b
a	d
b	c

## Fold/Unroll Rows

To pack or unpack the data into single value you can use **fold-by** and **unroll** functions.

**fold-by** groups dataset and packs columns data from each group separately into desired datastructure (like vector or sequence). **unroll** does the opposite.

### Fold-by

Group-by and pack columns into vector

```
(api/fold-by DS [:V3 :V4 :V1])
```

\_\_unnamed [6 4]:

:V4	:V3	:V1	:V2
B	1.000	1	[5]
C	1.500	2	[6]
C	1.500	1	[3 9]
A	0.5000	1	[1 7]
B	1.000	2	[2 8]
A	0.5000	2	[4]

You can pack several columns at once.

```
(api/fold-by DS [:V4])
```

\_\_unnamed [3 4]:

:V4	:V1	:V2	:V3
B	[2 1 2]	[2 5 8]	[1.0 1.0 1.0]
C	[1 2 1]	[3 6 9]	[1.5 1.5 1.5]
A	[1 2 1]	[1 4 7]	[0.5 0.5 0.5]

You can use custom packing function

```
(api/fold-by DS [:V4] seq)
```

\_\_unnamed [3 4]:

:V4	:V1	:V2	:V3
B	clojure.lang.LazySeq@7c02	clojure.lang.LazySeq@7c84	clojure.lang.LazySeq@1f0745f
C	clojure.lang.LazySeq@785f	clojure.lang.LazySeq@8065	clojure.lang.LazySeq@20f8745f
A	clojure.lang.LazySeq@785f	clojure.lang.LazySeq@78a3	clojure.lang.LazySeq@c3e0745f

or

```
(api/fold-by DS [:V4] set)
```

\_\_unnamed [3 4]:

:V4	:V1	:V2	:V3
B	#{1 2}	#{2 5 8}	#{1.0}
C	#{1 2}	#{6 3 9}	#{1.5}
A	#{1 2}	#{7 1 4}	#{0.5}

## Unroll

`unroll` unfolds sequences stored in data, multiplying other ones when necessary. You can unroll more than one column at once (folded data should have the same size!).

Options:

- `:indexes?` if true (or column name), information about index of unrolled sequence is added.
- `:datatypes` list of datatypes which should be applied to restored columns, a map

Unroll one column

```
(api/unroll (api/fold-by DS [:V4]) [:V1])
```

\_\_unnamed [9 4]:

:V4	:V2	:V3	:V1
B	[2 5 8]	[1.0 1.0 1.0]	2

	:V4	:V2	:V3	:V1
B		[2 5 8]	[1.0 1.0 1.0]	1
B		[2 5 8]	[1.0 1.0 1.0]	2
C		[3 6 9]	[1.5 1.5 1.5]	1
C		[3 6 9]	[1.5 1.5 1.5]	2
C		[3 6 9]	[1.5 1.5 1.5]	1
A		[1 4 7]	[0.5 0.5 0.5]	1
A		[1 4 7]	[0.5 0.5 0.5]	2
A		[1 4 7]	[0.5 0.5 0.5]	1

Unroll all folded columns

```
(api/unroll (api/fold-by DS [:V4]) [:V1 :V2 :V3])
```

\_\_unnamed [9 4]:

	:V4	:V1	:V2	:V3
B	2	2	1.000	
B	1	5	1.000	
B	2	8	1.000	
C	1	3	1.500	
C	2	6	1.500	
C	1	9	1.500	
A	1	1	0.5000	
A	2	4	0.5000	
A	1	7	0.5000	

Unroll one by one leads to cartesian product

```
(-> DS
  (api/fold-by [:V4 :V1])
  (api/unroll [:V2])
  (api/unroll [:V3]))
```

\_\_unnamed [15 4]:

	:V4	:V1	:V2	:V3
C	2	6	1.500	
A	1	1	0.5000	
A	1	1	0.5000	
A	1	7	0.5000	
A	1	7	0.5000	
B	1	5	1.000	
C	1	3	1.500	
C	1	3	1.500	
C	1	9	1.500	
C	1	9	1.500	
A	2	4	0.5000	
B	2	2	1.000	
B	2	2	1.000	

:V4	:V1	:V2	:V3
B	2	8	1.000
B	2	8	1.000

You can add indexes

```
(api/unroll (api/fold-by DS [:V1]) [:V4 :V2 :V3] {:indexes? true}))
```

\_\_unnamed [9 5]:

:V1	:indexes	:V2	:V3	:V4
1	0	1	0.5000	A
1	1	3	1.500	C
1	2	5	1.000	B
1	3	7	0.5000	A
1	4	9	1.500	C
2	0	2	1.000	B
2	1	4	0.5000	A
2	2	6	1.500	C
2	3	8	1.000	B

```
(api/unroll (api/fold-by DS [:V1]) [:V4 :V2 :V3] {:indexes? "vector idx"}))
```

\_\_unnamed [9 5]:

:V1	vector idx	:V2	:V3	:V4
1	0	1	0.5000	A
1	1	3	1.500	C
1	2	5	1.000	B
1	3	7	0.5000	A
1	4	9	1.500	C
2	0	2	1.000	B
2	1	4	0.5000	A
2	2	6	1.500	C
2	3	8	1.000	B

You can also force datatypes

```
(-> DS
  (api/fold-by [:V1])
  (api/unroll [:V4 :V2 :V3] {:datatypes {:V4 :string
                                          :V2 :int16
                                          :V3 :float32}})
  (api/info :columns))
```

\_\_unnamed :column info [4 4]:

:name	:size	:datatype	:categorical?
:V1	9	:object	

:name	:size	:datatype	:categorical?
:V2	9	:int16	
:V3	9	:float32	
:V4	9	:string	true

**Reshape**

**Longer**

**Wider**

**Join/Concat**

**Left**

**Right**

**Inner**

**Hash**

**Concat**

**Functions**