Proposed API for tech.ml.dataset

GenerateMe

2020-05-27

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
https://vega.github.io/vega-lite/examples/data/seattle-weather.csv	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, 3.20
#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, 2012-0
```

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, 2012-0
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, 5.3
```

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 0x359867fa "2012-01-01"] 0.0 12.8 5.0 4.7 "drizzle"] [#object[java.time.L
```

Rows as sequence of maps

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

```
({"date" #object[java.time.LocalDate 0x798121af "2012-01-01"],
  "precipitation" 0.0,
  "temp_min" 5.0,
```

```
  "weather" "drizzle",
  "temp_max" 12.8,
  "wind" 4.7}
 {"date" #object[java.time.LocalDate 0x3eec2d30 "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]: | :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 |

```
(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 gruping 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 concats 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 supress 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 ungrouping 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 cat 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 operated 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 colums 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@1dc5451b
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@3d83ea0b
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@3d83ea0b
2	2	1.000	B
2	4	0.5000	A
2	6	1.500	C

v1	v2	[1 2 3]	java.lang.Object@3d83ea0b
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
0.2565	1	0.5000	A
0.7238	2	1.000	B
0.7878	3	1.500	C
0.1993	4	0.5000	A
0.8512	5	1.000	B
0.5382	6	1.500	C

:V1	:V2	:V3	:V4
0.4285	7	0.5000	A
0.4484	8	1.000	B
0.1991	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	

Tha 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

Let's add numerical columns together

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

_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)) (api/column-names DS  #{:int64 :float64} :datat
    (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
1	5	1.000	B	7.000

:V1	:V2	:V3	:V4	:sum-of-numbers
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 unseleted 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 infomation 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 haxadecimal 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	0.5000	10
2	2	1.000	11
1	3	1.500	12
2	4	0.5000	10
1	5	1.000	11
2	6	1.500	12
1	7	0.5000	10
2	8	1.000	11
1	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" 0x44222d1 "[J@44222d1"]
```

Function also works on grouped dataset

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

```
(#object["[J" 0x15a01a22 "[J@15a01a22"] #object["[J" 0x53e15661 "[J@53e15661"] #object["[J" 0x2412fe45 "
```

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;" 0x4a851c75 "[Ljava.lang.String;@4a851c75"]
#object["[F" 0x5181d68e "[F@5181d68e"]
```

Rows

Rows can be selected or dropped using various selectors:

- row id(s) - row index as number or seqence 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
1	9	1.500	C

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

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

_unnamed [9 4]:

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

Five random rows with repetition

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

_unnamed [5 4]:

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

Five random, non-repeating rows

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

_unnamed [5 4]:

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

Shuffle dataset

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

_unnamed [9 4]:

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

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
2	4	0.5000	A

:V1	:V2	:V3	:V4
1	1	0.5000	A
1	1	0.5000	A
2	4	0.5000	A
1	1	0.5000	A
1	5	1.000	B
2	2	1.000	B
1	5	1.000	B
2	2	1.000	B
1	5	1.000	B
2	6	1.500	C
1	9	1.500	C
2	6	1.500	C
1	9	1.500	C
1	9	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 aggreation. This can be turned of 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 stetting `: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 provied 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 achived 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	8	1.000	B
1	9	1.500	C

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 seqence 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 :fract-part] (fn [^double v]
                                                       [(int (quot v 1.0))
                                                        (mod v 1.0)]))
    (api/ungroup))
```

_unnamed [9 5]:

:V1	:V2	:int-part	:fract-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}

This works also on grouped dataset

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

_unnamed [6 4]:

:V4	:V1	:V2	:V3
B	[1]	[5]	[1.0]
C	[1 1]	[3 9]	[1.5 1.5]
A	[1 1]	[1 7]	[0.5 0.5]
B	[2 2]	[2 8]	[1.0 1.0]
C	[2]	[6]	[1.5]
A	[2]	[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
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
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	
:V2	9	:int16	
:V3	9	:float32	
:V4	9	:string	true

This works also on grouped dataset

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

_unnamed [9 5]:

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

Reshape

Reshaping data provides two types of operations:

- `pivot->longer` - converting columns to rows
- `pivot->wider` - converting rows to columns

Both functions are inspired on tidyr R package and provide almost the same functionality.

All examples are taken from mentioned above documentation.

Both functions work only on regular dataset.

Longer

`pivot->longer` converts columns to rows. Column names are treated as data.

Arguments:

- dataset
- columns selector
- options:
  - `:target-columns` - column name(s) where source column names are stored, or columns pattern (see below) (default: `:$column`)
  - `:value-column-name` - name of the column for values (default: `:$value`)
  - `:splitter` - regular expression or function which splits source column names into data
  - `:drop-missing?` - remove rows with missing? (default: `:true`)
  - `:datatypes` - map of target columns data types

`:target-columns` - can be:

- column name - source columns names are put there as a data
- column names as seqence - source columns names after split are put separately into `:target-columns` as data
- pattern - is a sequence of names, where some of the names are `nil`. `nil` is replaced by a name taken from splitter and such column is used for values.

Create rows from all columns but `"religion"`.

```
(def relig-income (api/dataset "data/relig_income.csv"))
```

```
(api/head relig-income)
```

data/relig_income.csv [5 11]:

religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k	$50-75k	$75-100k	$100-150k	>150k	Don't know/refused
Agnostic	27	34	60	81	76	137	122	109	84	96
Atheist	12	27	37	52	35	70	73	59	74	76
Buddhist	27	21	30	34	33	58	62	39	53	54
Catholic	418	617	732	670	638	1116	949	792	633	1489
Don't know/refused	15	14	15	11	10	35	21	17	18	116

```
(api/pivot->longer relig-income (complement #{"religion"}))
```

data/relig_income.csv [180 3]:

religion	:$column	:$value
Agnostic	<$10k	27
Atheist	<$10k	12
Buddhist	<$10k	27
Catholic	<$10k	418
Don't know/refused	<$10k	15
Evangelical Prot	<$10k	575
Hindu	<$10k	1
Historically Black Prot	<$10k	228
Jehovah's Witness	<$10k	20
Jewish	<$10k	19

religion	:$column	:$value
Mainline Prot	<$10k	289
Mormon	<$10k	29
Muslim	<$10k	6
Orthodox	<$10k	13
Other Christian	<$10k	9
Other Faiths	<$10k	20
Other World Religions	<$10k	5
Unaffiliated	<$10k	217
Agnostic	Don't know/refused	96
Atheist	Don't know/refused	76
Buddhist	Don't know/refused	54
Catholic	Don't know/refused	1489
Don't know/refused	Don't know/refused	116
Evangelical Prot	Don't know/refused	1529
Hindu	Don't know/refused	37

Convert only columns starting with `"wk"` and pack them into `:week` column, values go to `:rank` column

```
(def bilboard (-> (api/dataset "data/billboard.csv.gz")
                  (api/drop-columns #(= :boolean %) :datatype))) ;; drop some boolean columns, tidyr ju
```

```
(->> bilboard
     (api/column-names)
     (take 13)
     (api/select-columns bilboard)
     (api/head))
```

data/billboard.csv.gz [5 13]:

artist	track	date.entered	wk1	wk2	wk3	wk4	wk5	wk6	wk7	wk8	wk9	wk1
2 Pac	Baby Don't Cry (Keep…	2000-02-26	87	82	72	77	87	94	99			
2Ge+her	The Hardest Part Of …	2000-09-02	91	87	92							
3 Doors Down	Kryptonite	2000-04-08	81	70	68	67	66	57	54	53	51	51
3 Doors Down	Loser	2000-10-21	76	76	72	69	67	65	55	59	62	61
504 Boyz	Wobble Wobble	2000-04-15	57	34	25	17	17	31	36	49	53	57

```
(api/pivot->longer bilboard #(clojure.string/starts-with? % "wk") {:target-columns :week
                                                                   :value-column-name :rank})
```

data/billboard.csv.gz [5307 5]:

artist	track	date.entered	:week	:rank
3 Doors Down	Kryptonite	2000-04-08	wk35	4
Braxton, Toni	He Wasn't Man Enough	2000-03-18	wk35	34
Creed	Higher	1999-09-11	wk35	22
Creed	With Arms Wide Open	2000-05-13	wk35	5
Hill, Faith	Breathe	1999-11-06	wk35	8
Joe	I Wanna Know	2000-01-01	wk35	5
Lonestar	Amazed	1999-06-05	wk35	14
Vertical Horizon	Everything You Want	2000-01-22	wk35	27

artist	track	date.entered	:week	:rank
matchbox twenty	Bent	2000-04-29	wk35	33
Creed	Higher	1999-09-11	wk55	21
Lonestar	Amazed	1999-06-05	wk55	22
3 Doors Down	Kryptonite	2000-04-08	wk19	18
3 Doors Down	Loser	2000-10-21	wk19	73
98^0	Give Me Just One Nig…	2000-08-19	wk19	93
Aaliyah	I Don't Wanna	2000-01-29	wk19	83
Aaliyah	Try Again	2000-03-18	wk19	3
Adams, Yolanda	Open My Heart	2000-08-26	wk19	79
Aguilera, Christina	Come On Over Baby (A…	2000-08-05	wk19	23
Aguilera, Christina	I Turn To You	2000-04-15	wk19	29
Aguilera, Christina	What A Girl Wants	1999-11-27	wk19	18
Alice Deejay	Better Off Alone	2000-04-08	wk19	79
Amber	Sexual	1999-07-17	wk19	95
Anthony, Marc	My Baby You	2000-09-16	wk19	91
Anthony, Marc	You Sang To Me	2000-02-26	wk19	9
Avant	My First Love	2000-11-04	wk19	81

We can create numerical column out of column names

```
(api/pivot->longer bilboard #(clojure.string/starts-with? % "wk") {:target-columns :week
                                                                   :value-column-name :rank
                                                                   :splitter #"wk(.*)"
                                                                   :datatypes {:week :int16}})
```

data/billboard.csv.gz [5307 5]:

artist	track	date.entered	:week	:rank
3 Doors Down	Kryptonite	2000-04-08	46	21
Creed	Higher	1999-09-11	46	7
Creed	With Arms Wide Open	2000-05-13	46	37
Hill, Faith	Breathe	1999-11-06	46	31
Lonestar	Amazed	1999-06-05	46	5
3 Doors Down	Kryptonite	2000-04-08	51	42
Creed	Higher	1999-09-11	51	14
Hill, Faith	Breathe	1999-11-06	51	49
Lonestar	Amazed	1999-06-05	51	12
2 Pac	Baby Don't Cry (Keep…	2000-02-26	6	94
3 Doors Down	Kryptonite	2000-04-08	6	57
3 Doors Down	Loser	2000-10-21	6	65
504 Boyz	Wobble Wobble	2000-04-15	6	31
98^0	Give Me Just One Nig…	2000-08-19	6	19
Aaliyah	I Don't Wanna	2000-01-29	6	35
Aaliyah	Try Again	2000-03-18	6	18
Adams, Yolanda	Open My Heart	2000-08-26	6	67
Adkins, Trace	More	2000-04-29	6	69
Aguilera, Christina	Come On Over Baby (A…	2000-08-05	6	18
Aguilera, Christina	I Turn To You	2000-04-15	6	19
Aguilera, Christina	What A Girl Wants	1999-11-27	6	13
Alice Deejay	Better Off Alone	2000-04-08	6	36
Amber	Sexual	1999-07-17	6	93

artist	track	date.entered	:week	:rank
Anthony, Marc	My Baby You	2000-09-16	6	81
Anthony, Marc	You Sang To Me	2000-02-26	6	27

When column names contain observation data, such column names can be splitted and data can be restored into separate columns.

```
(def who (api/dataset "data/who.csv.gz"))
```

```
(->> who
     (api/column-names)
     (take 10)
     (api/select-columns who)
     (api/head))
```

data/who.csv.gz [5 10]:

country	iso2	iso3	year	new_sp_m014	new_sp_m1524	new_sp_m2534	new_sp_m3544	new_sp_m4554	new_sp_m5564
Afghanistan	AF	AFG	1980						
Afghanistan	AF	AFG	1981						
Afghanistan	AF	AFG	1982						
Afghanistan	AF	AFG	1983						
Afghanistan	AF	AFG	1984						

```
(api/pivot->longer who #(clojure.string/starts-with? % "new") {:target-columns [:diagnosis :gender :age]
                                                               :splitter #"new_?(.*)_(.)(.*)"
                                                               :value-column-name :count})
```

data/who.csv.gz [76046 8]:

country	iso2	iso3	year	:diagnosis	:gender	:age	:count
Albania	AL	ALB	2013	rel	m	1524	60
Algeria	DZ	DZA	2013	rel	m	1524	1021
Andorra	AD	AND	2013	rel	m	1524	0
Angola	AO	AGO	2013	rel	m	1524	2992
Anguilla	AI	AIA	2013	rel	m	1524	0
Antigua and Barbuda	AG	ATG	2013	rel	m	1524	1
Argentina	AR	ARG	2013	rel	m	1524	1124
Armenia	AM	ARM	2013	rel	m	1524	116
Australia	AU	AUS	2013	rel	m	1524	105
Austria	AT	AUT	2013	rel	m	1524	44
Azerbaijan	AZ	AZE	2013	rel	m	1524	958
Bahamas	BS	BHS	2013	rel	m	1524	2
Bahrain	BH	BHR	2013	rel	m	1524	13
Bangladesh	BD	BGD	2013	rel	m	1524	14705
Barbados	BB	BRB	2013	rel	m	1524	0
Belarus	BY	BLR	2013	rel	m	1524	162
Belgium	BE	BEL	2013	rel	m	1524	63
Belize	BZ	BLZ	2013	rel	m	1524	8
Benin	BJ	BEN	2013	rel	m	1524	301
Bermuda	BM	BMU	2013	rel	m	1524	0

country	iso2	iso3	year	:diagnosis	:gender	:age	:count
Bhutan	BT	BTN	2013	rel	m	1524	180
Bolivia (Plurinational State of)	BO	BOL	2013	rel	m	1524	1470
Bonaire, Saint Eustatius and Saba	BQ	BES	2013	rel	m	1524	0
Bosnia and Herzegovina	BA	BIH	2013	rel	m	1524	57
Botswana	BW	BWA	2013	rel	m	1524	423

When data contains multiple observations per row, we can use splitter and pattern for target columns to create new columns and put values there. In following dataset we have two obseravations `dob` and `gender` for two childs. We want to put child infomation into the column and leave dob and gender for values.

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

```
family
```

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

```
(api/pivot->longer family (complement #{"family"}) {:target-columns [nil :child]
                                                    :splitter #(clojure.string/split % #"_")
                                                    :datatypes {"gender" :int16}})
```

data/family.csv [9 4]:

family	:child	dob	gender
1	child1	1998-11-26	1
2	child1	1996-06-22	2
3	child1	2002-07-11	2
4	child1	2004-10-10	1
5	child1	2000-12-05	2
1	child2	2000-01-29	2
3	child2	2004-04-05	2
4	child2	2009-08-27	1
5	child2	2005-02-28	1

Similar here, we have two observations: `x` and `y` in four groups.

```
(def anscombe (api/dataset "data/anscombe.csv"))
```

```
anscombe
```

data/anscombe.csv [11 8]:

x1	x2	x3	x4	y1	y2	y3	y4
10	10	10	8	8.040	9.140	7.460	6.580
8	8	8	8	6.950	8.140	6.770	5.760
13	13	13	8	7.580	8.740	12.74	7.710
9	9	9	8	8.810	8.770	7.110	8.840
11	11	11	8	8.330	9.260	7.810	8.470
14	14	14	8	9.960	8.100	8.840	7.040
6	6	6	8	7.240	6.130	6.080	5.250
4	4	4	19	4.260	3.100	5.390	12.50
12	12	12	8	10.84	9.130	8.150	5.560
7	7	7	8	4.820	7.260	6.420	7.910
5	5	5	8	5.680	4.740	5.730	6.890

```
(api/pivot->longer anscombe :all {:splitter #"(.)(.)"
                                  :target-columns [nil :set]})
```

data/anscombe.csv [44 3]:

:set	x	y
1	10	8.040
1	8	6.950
1	13	7.580
1	9	8.810
1	11	8.330
1	14	9.960
1	6	7.240
1	4	4.260
1	12	10.84
1	7	4.820
1	5	5.680
2	10	9.140
2	8	8.140
2	13	8.740
2	9	8.770
2	11	9.260
2	14	8.100
2	6	6.130
2	4	3.100
2	12	9.130
2	7	7.260
2	5	4.740
3	10	7.460
3	8	6.770
3	13	12.74

```
(def pnl (api/dataset {:x [1 2 3 4]
                       :a [1 1 0 0]
                       :b [0 1 1 1]
                       :y1 (repeatedly 4 rand)
                       :y2 (repeatedly 4 rand)
```

```
                        :z1 [3 3 3 3]
                        :z2 [-2 -2 -2 -2]}))
```

```
pnl
```

_unnamed [4 7]:

:x	:a	:b	:y1	:y2	:z1	:z2
1	1	0	0.3789	0.9831	3	-2
2	1	1	0.3252	0.9221	3	-2
3	0	1	0.3689	0.6474	3	-2
4	0	1	0.7811	0.8229	3	-2

```
(api/pivot->longer pnl [:y1 :y2 :z1 :z2] {:target-columns [nil :times]
                                          :splitter #":(.)(.)"})
```

_unnamed [8 6]:

:x	:a	:b	:times	y	z
1	1	0	1	0.3789	3
2	1	1	1	0.3252	3
3	0	1	1	0.3689	3
4	0	1	1	0.7811	3
1	1	0	2	0.9831	-2
2	1	1	2	0.9221	-2
3	0	1	2	0.6474	-2
4	0	1	2	0.8229	-2

Wider

`pivot->wider` converts rows to columns.

Arguments:

- dataset
- columns selector - values from selected columns are converted to new columns
- value columns - what are values

When multiple columns are used as columns selector, names are joined using `:separator` (default: "_") option.

When columns selector creates non unique set of values, they are folded using `:fold-fn` (default: `vec`) option.

When value columns are a sequence, multiple observations as columns are created appending value column names into new columns. Column names are joined using `:value-separator` (default: "-") option.

Use `station` as a name source for columns and `seen` for values

```
(def fish (api/dataset "data/fish_encounters.csv"))
```

```
fish
```

data/fish_encounters.csv [114 3]:

fish	station	seen
4842	Release	1
4842	I80_1	1
4842	Lisbon	1
4842	Rstr	1
4842	Base_TD	1
4842	BCE	1
4842	BCW	1
4842	BCE2	1
4842	BCW2	1
4842	MAE	1
4842	MAW	1
4843	Release	1
4843	I80_1	1
4843	Lisbon	1
4843	Rstr	1
4843	Base_TD	1
4843	BCE	1
4843	BCW	1
4843	BCE2	1
4843	BCW2	1
4843	MAE	1
4843	MAW	1
4844	Release	1
4844	I80_1	1
4844	Lisbon	1

```
(api/pivot->wider fish "station" "seen")
```

data/fish_encounters.csv [19 12]:

fish	Rstr	Base_TD	I80_1	Release	MAE	BCE2	MAW	BCW2	BCE	Lisbon	BCW
4842	1	1	1	1	1	1	1	1	1	1	1
4843	1	1	1	1	1	1	1	1	1	1	1
4844	1	1	1	1	1	1	1	1	1	1	1
4850	1	1	1	1					1		1
4857	1	1	1	1		1		1	1	1	1
4858	1	1	1	1	1	1	1	1	1	1	1
4861	1	1	1	1	1	1	1	1	1	1	1
4862	1	1	1	1		1		1	1	1	1
4864			1	1							
4865			1	1						1	
4845	1	1	1	1						1	
4847			1	1						1	
4848	1		1	1						1	
4849			1	1							
4851			1	1							
4854			1	1							
4855	1	1	1	1						1	
4859	1	1	1	1						1	
4863			1	1							

If selected columns contain multiple values, such values should be folded.

```
(def warpbreaks (api/dataset "data/warpbreaks.csv"))
```

```
warpbreaks
```

data/warpbreaks.csv [54 3]:

breaks	wool	tension
26	A	L
30	A	L
54	A	L
25	A	L
70	A	L
52	A	L
51	A	L
26	A	L
67	A	L
18	A	M
21	A	M
29	A	M
17	A	M
12	A	M
18	A	M
35	A	M
30	A	M
36	A	M
36	A	H
21	A	H
24	A	H
18	A	H
10	A	H
43	A	H
28	A	H

Let's see how many values are for each type of `wool` and `tension` groups

```
(-> warpbreaks
    (api/group-by ["wool" "tension"])
    (api/aggregate {:n api/row-count}))
```

_unnamed [6 3]:

wool	tension	:n
A	H	9
B	H	9
A	L	9
A	M	9
B	L	9
B	M	9

```
(-> warpbreaks
    (api/reorder-columns ["wool" "tension" "breaks"])
    (api/pivot->wider "wool" "breaks" {:fold-fn vec}))
```

data/warpbreaks.csv [3 3]:

tension	B	A
M	[42 26 19 16 39 28 21 39 29]	[18 21 29 17 12 18 35 30 36]
H	[20 21 24 17 13 15 15 16 28]	[36 21 24 18 10 43 28 15 26]
L	[27 14 29 19 29 31 41 20 44]	[26 30 54 25 70 52 51 26 67]

We can also calculate mean (aggreate values)

```
(-> warpbreaks
    (api/reorder-columns ["wool" "tension" "breaks"])
    (api/pivot->wider "wool" "breaks" {:fold-fn tech.v2.datatype.functional/mean}))
```

data/warpbreaks.csv [3 3]:

tension	B	A
H	18.78	24.56
M	28.78	24.00
L	28.22	44.56

Multiple source columns, joined with default separator.

```
(def production (api/dataset "data/production.csv"))
```

```
production
```

data/production.csv [45 4]:

product	country	year	production
A	AI	2000	1.637
A	AI	2001	0.1587
A	AI	2002	-1.568
A	AI	2003	-0.4446
A	AI	2004	-0.07134
A	AI	2005	1.612
A	AI	2006	-0.7043
A	AI	2007	-1.536
A	AI	2008	0.8391
A	AI	2009	-0.3742
A	AI	2010	-0.7116
A	AI	2011	1.128
A	AI	2012	1.457
A	AI	2013	-1.559
A	AI	2014	-0.1170
B	AI	2000	-0.02618
B	AI	2001	-0.6886
B	AI	2002	0.06249

product	country	year	production
B	AI	2003	-0.7234
B	AI	2004	0.4725
B	AI	2005	-0.9417
B	AI	2006	-0.3478
B	AI	2007	0.5243
B	AI	2008	1.832
B	AI	2009	0.1071

```
(api/pivot->wider production ["product" "country"] "production")
```

data/production.csv [15 4]:

year	A_AI	B_EI	B_AI
2000	1.637	1.405	-0.02618
2001	0.1587	-0.5962	-0.6886
2002	-1.568	-0.2657	0.06249
2003	-0.4446	0.6526	-0.7234
2004	-0.07134	0.6256	0.4725
2005	1.612	-1.345	-0.9417
2006	-0.7043	-0.9718	-0.3478
2007	-1.536	-1.697	0.5243
2008	0.8391	0.04556	1.832
2009	-0.3742	1.193	0.1071
2010	-0.7116	-1.606	-0.3290
2011	1.128	-0.7724	-1.783
2012	1.457	-2.503	0.6113
2013	-1.559	-1.628	-0.7853
2014	-0.1170	0.03330	0.9784

Multiple value columns

```
(def income (api/dataset "data/us_rent_income.csv"))
```

```
income
```

data/us_rent_income.csv [104 5]:

GEOID	NAME	variable	estimate	moe
1	Alabama	income	24476	136
1	Alabama	rent	747	3
2	Alaska	income	32940	508
2	Alaska	rent	1200	13
4	Arizona	income	27517	148
4	Arizona	rent	972	4
5	Arkansas	income	23789	165
5	Arkansas	rent	709	5
6	California	income	29454	109
6	California	rent	1358	3
8	Colorado	income	32401	109
8	Colorado	rent	1125	5

GEOID	NAME	variable	estimate	moe
9	Connecticut	income	35326	195
9	Connecticut	rent	1123	5
10	Delaware	income	31560	247
10	Delaware	rent	1076	10
11	District of Columbia	income	43198	681
11	District of Columbia	rent	1424	17
12	Florida	income	25952	70
12	Florida	rent	1077	3
13	Georgia	income	27024	106
13	Georgia	rent	927	3
15	Hawaii	income	32453	218
15	Hawaii	rent	1507	18
16	Idaho	income	25298	208

```
(api/pivot->wider income "variable" ["estimate" "moe"])
```

data/us_rent_income.csv [52 6]:

GEOID	NAME	estimate-rent	moe-rent	estimate-income	moe-income
1	Alabama	747	3	24476	136
2	Alaska	1200	13	32940	508
4	Arizona	972	4	27517	148
5	Arkansas	709	5	23789	165
6	California	1358	3	29454	109
8	Colorado	1125	5	32401	109
9	Connecticut	1123	5	35326	195
10	Delaware	1076	10	31560	247
11	District of Columbia	1424	17	43198	681
12	Florida	1077	3	25952	70
13	Georgia	927	3	27024	106
15	Hawaii	1507	18	32453	218
16	Idaho	792	7	25298	208
17	Illinois	952	3	30684	83
18	Indiana	782	3	27247	117
19	Iowa	740	4	30002	143
20	Kansas	801	5	29126	208
21	Kentucky	713	4	24702	159
22	Louisiana	825	4	25086	155
23	Maine	808	7	26841	187
24	Maryland	1311	5	37147	152
25	Massachusetts	1173	5	34498	199
26	Michigan	824	3	26987	82
27	Minnesota	906	4	32734	189
28	Mississippi	740	5	22766	194

Reshape contact data

```
(def contacts (api/dataset "data/contacts.csv"))
```

```
contacts
```

data/contacts.csv [6 3]:

field	value	person_id
name	Jiena McLellan	1
company	Toyota	1
name	John Smith	2
company	google	2
email	john@google.com	2
name	Huxley Ratcliffe	3

```
(api/pivot->wider contacts "field" "value")
```

data/contacts.csv [3 4]:

person_id	email	name	company
1		Jiena McLellan	Toyota
2	john@google.com	John Smith	google
3		Huxley Ratcliffe	

Reshaping

A couple of `tidyr` examples of more complex reshaping.

World bank

```
(def world-bank-pop (api/dataset "data/world_bank_pop.csv.gz"))
```

```
(->> world-bank-pop
     (api/column-names)
     (take 8)
     (api/select-columns world-bank-pop)
     (api/head))
```

data/world_bank_pop.csv.gz [5 8]:

country	indicator	2000	2001	2002	2003	2004	2005
ABW	SP.URB.TOTL	4.244E+04	4.305E+04	4.367E+04	4.425E+04	4.467E+04	4.489E+04
ABW	SP.URB.GROW	1.183	1.413	1.435	1.310	0.9515	0.4913
ABW	SP.POP.TOTL	9.085E+04	9.290E+04	9.499E+04	9.702E+04	9.874E+04	1.000E+05
ABW	SP.POP.GROW	2.055	2.226	2.229	2.109	1.757	1.302
AFG	SP.URB.TOTL	4.436E+06	4.648E+06	4.893E+06	5.156E+06	5.427E+06	5.692E+06

Step 1 - convert years column into values

```
(def pop2 (api/pivot->longer world-bank-pop (map str (range 2000 2018)) {:drop-missing? false
                                                                          :target-columns ["year"]
                                                                          :value-column-name "value"}))
```

```
pop2
```

data/world_bank_pop.csv.gz [19008 4]:

country	indicator	year	value
ABW	SP.URB.TOTL	2013	4.436E+04
ABW	SP.URB.GROW	2013	0.6695
ABW	SP.POP.TOTL	2013	1.032E+05
ABW	SP.POP.GROW	2013	0.5929
AFG	SP.URB.TOTL	2013	7.734E+06
AFG	SP.URB.GROW	2013	4.193
AFG	SP.POP.TOTL	2013	3.173E+07
AFG	SP.POP.GROW	2013	3.315
AGO	SP.URB.TOTL	2013	1.612E+07
AGO	SP.URB.GROW	2013	4.723
AGO	SP.POP.TOTL	2013	2.600E+07
AGO	SP.POP.GROW	2013	3.532
ALB	SP.URB.TOTL	2013	1.604E+06
ALB	SP.URB.GROW	2013	1.744
ALB	SP.POP.TOTL	2013	2.895E+06
ALB	SP.POP.GROW	2013	-0.1832
AND	SP.URB.TOTL	2013	7.153E+04
AND	SP.URB.GROW	2013	-2.119
AND	SP.POP.TOTL	2013	8.079E+04
AND	SP.POP.GROW	2013	-2.013
ARB	SP.URB.TOTL	2013	2.186E+08
ARB	SP.URB.GROW	2013	2.783
ARB	SP.POP.TOTL	2013	3.817E+08
ARB	SP.POP.GROW	2013	2.249
ARE	SP.URB.TOTL	2013	7.661E+06

Step 2 - separate `"indicate"` column

```
(def pop3 (api/separate-column pop2
                               "indicator" ["area" "variable"]
                               #(rest (clojure.string/split % #"\."))))
```

```
pop3
```

data/world_bank_pop.csv.gz [19008 5]:

country	area	variable	year	value
ABW	URB	TOTL	2013	4.436E+04
ABW	URB	GROW	2013	0.6695
ABW	POP	TOTL	2013	1.032E+05
ABW	POP	GROW	2013	0.5929
AFG	URB	TOTL	2013	7.734E+06
AFG	URB	GROW	2013	4.193
AFG	POP	TOTL	2013	3.173E+07
AFG	POP	GROW	2013	3.315
AGO	URB	TOTL	2013	1.612E+07
AGO	URB	GROW	2013	4.723
AGO	POP	TOTL	2013	2.600E+07

country	area	variable	year	value
AGO	POP	GROW	2013	3.532
ALB	URB	TOTL	2013	1.604E+06
ALB	URB	GROW	2013	1.744
ALB	POP	TOTL	2013	2.895E+06
ALB	POP	GROW	2013	-0.1832
AND	URB	TOTL	2013	7.153E+04
AND	URB	GROW	2013	-2.119
AND	POP	TOTL	2013	8.079E+04
AND	POP	GROW	2013	-2.013
ARB	URB	TOTL	2013	2.186E+08
ARB	URB	GROW	2013	2.783
ARB	POP	TOTL	2013	3.817E+08
ARB	POP	GROW	2013	2.249
ARE	URB	TOTL	2013	7.661E+06

Step 3 - Make columns based on `"variable"` values.

```
(api/pivot->wider pop3 "variable" "value")
```

data/world_bank_pop.csv.gz [9504 5]:

country	area	year	GROW	TOTL
ABW	URB	2013	0.6695	4.436E+04
ABW	POP	2013	0.5929	1.032E+05
AFG	URB	2013	4.193	7.734E+06
AFG	POP	2013	3.315	3.173E+07
AGO	URB	2013	4.723	1.612E+07
AGO	POP	2013	3.532	2.600E+07
ALB	URB	2013	1.744	1.604E+06
ALB	POP	2013	-0.1832	2.895E+06
AND	URB	2013	-2.119	7.153E+04
AND	POP	2013	-2.013	8.079E+04
ARB	URB	2013	2.783	2.186E+08
ARB	POP	2013	2.249	3.817E+08
ARE	URB	2013	1.555	7.661E+06
ARE	POP	2013	1.182	9.006E+06
ARG	URB	2013	1.188	3.882E+07
ARG	POP	2013	1.047	4.254E+07
ARM	URB	2013	0.2810	1.828E+06
ARM	POP	2013	0.4013	2.894E+06
ASM	URB	2013	0.05798	4.831E+04
ASM	POP	2013	0.1393	5.531E+04
ATG	URB	2013	0.3838	2.480E+04
ATG	POP	2013	1.076	9.782E+04
AUS	URB	2013	1.875	1.979E+07
AUS	POP	2013	1.758	2.315E+07
AUT	URB	2013	0.9196	4.862E+06

Multi-choice

```
(def multi (api/dataset {:id [1 2 3 4]
                         :choice1 ["A" "C" "D" "B"]
                         :choice2 ["B" "B" nil "D"]
                         :choice3 ["C" nil nil nil]}))
```

```
multi
```

_unnamed [4 4]:

:id	:choice1	:choice2	:choice3
1	A	B	C
2	C	B	
3	D		
4	B	D	

Step 1 - convert all choices into rows and add artificial column to all values which are not missing.

```
(def multi2 (-> multi
                (api/pivot->longer (complement #{:id}))
                (api/add-or-update-column :checked true)))
```

```
multi2
```

_unnamed [8 4]:

:id	:$column	:$value	:checked
1	:choice1	A	true
2	:choice1	C	true
3	:choice1	D	true
4	:choice1	B	true
1	:choice2	B	true
2	:choice2	B	true
4	:choice2	D	true
1	:choice3	C	true

Step 2 - Convert back to wide form with actual choices as columns

```
(-> multi2
    (api/drop-columns :$column)
    (api/pivot->wider :$value :checked {:drop-missing? false})
    (api/order-by :id))
```

_unnamed [4 5]:

:id	A	B	C	D
1	true	true	true	
2		true	true	
3				true
4		true		true

Construction

```
(def construction (api/dataset "data/construction.csv"))
(def construction-unit-map {"1 unit" "1"
                            "2 to 4 units" "2-4"
                            "5 units or more" "5+"})
```

```
construction
```

data/construction.csv [9 9]:

Year	Month	1 unit	2 to 4 units	5 units or more	Northeast	Midwest	South	West
2018	January	859		348	114	169	596	339
2018	February	882		400	138	160	655	336
2018	March	862		356	150	154	595	330
2018	April	797		447	144	196	613	304
2018	May	875		364	90	169	673	319
2018	June	867		342	76	170	610	360
2018	July	829		360	108	183	594	310
2018	August	939		286	90	205	649	286
2018	September	835		304	117	175	560	296

Conversion 1 - Group two column types

```
(-> construction
    (api/pivot->longer #"^[125NWS].*|Midwest" {:target-columns [:units :region]
                                               :splitter (fn [col-name]
                                                           (if (re-matches #"^[125].*" col-name)
                                                             [(construction-unit-map col-name) nil]
                                                             [nil col-name]))
                                               :value-column-name :n
                                               :drop-missing? false}))
```

data/construction.csv [63 5]:

Year	Month	:units	:region	:n
2018	January	1		859
2018	February	1		882
2018	March	1		862
2018	April	1		797
2018	May	1		875
2018	June	1		867
2018	July	1		829
2018	August	1		939
2018	September	1		835
2018	January	2-4		
2018	February	2-4		
2018	March	2-4		
2018	April	2-4		
2018	May	2-4		
2018	June	2-4		
2018	July	2-4		
2018	August	2-4		

Year	Month	:units	:region	:n
2018	September	2-4		
2018	January	5+		348
2018	February	5+		400
2018	March	5+		356
2018	April	5+		447
2018	May	5+		364
2018	June	5+		342
2018	July	5+		360

Conversion 2 - Convert to longer form and back and rename columns

```
(-> construction
    (api/pivot->longer #"^[125NWS].*|Midwest" {:target-columns [:units :region]
                                               :splitter (fn [col-name]
                                                           (if (re-matches #"^[125].*" col-name)
                                                             [(construction-unit-map col-name) nil]
                                                             [nil col-name]))
                                               :value-column-name :n
                                               :drop-missing? false})
    (api/pivot->wider [:units :region] :n)
    (api/rename-columns (zipmap (vals construction-unit-map)
                                (keys construction-unit-map))))
```

data/construction.csv [9 9]:

Year	Month	Midwest	5 units or more	2 to 4 units	Northeast	South	1 unit	West
2018	January	169	348		114	596	859	339
2018	February	160	400		138	655	882	336
2018	March	154	356		150	595	862	330
2018	April	196	447		144	613	797	304
2018	May	169	364		90	673	875	319
2018	June	170	342		76	610	867	360
2018	July	183	360		108	594	829	310
2018	August	205	286		90	649	939	286
2018	September	175	304		117	560	835	296

Various operations on stocks, examples taken from gather and spread manuals.

```
(def stocks-tidyr (api/dataset "data/stockstidyr.csv"))
```

```
stocks-tidyr
```

data/stockstidyr.csv [10 4]:

time	X	Y	Z
2009-01-01	1.310	-1.890	-1.779
2009-01-02	-0.2999	-1.825	2.399
2009-01-03	0.5365	-1.036	-3.987
2009-01-04	-1.884	-0.5218	-2.831

time	X	Y	Z
2009-01-05	-0.9605	-2.217	1.437
2009-01-06	-1.185	-2.894	3.398
2009-01-07	-0.8521	-2.168	-1.201
2009-01-08	0.2523	-0.3285	-1.532
2009-01-09	0.4026	1.964	-6.809
2009-01-10	-0.6438	2.686	-2.559

Convert to longer form

```
(def stocks-long (api/pivot->longer stocks-tidyr ["X" "Y" "Z"] {:value-column-name :price
                                                                :target-columns :stocks}))
```

```
stocks-long
```

data/stockstidyr.csv [30 3]:

time	:stocks	:price
2009-01-01	X	1.310
2009-01-02	X	-0.2999
2009-01-03	X	0.5365
2009-01-04	X	-1.884
2009-01-05	X	-0.9605
2009-01-06	X	-1.185
2009-01-07	X	-0.8521
2009-01-08	X	0.2523
2009-01-09	X	0.4026
2009-01-10	X	-0.6438
2009-01-01	Y	-1.890
2009-01-02	Y	-1.825
2009-01-03	Y	-1.036
2009-01-04	Y	-0.5218
2009-01-05	Y	-2.217
2009-01-06	Y	-2.894
2009-01-07	Y	-2.168
2009-01-08	Y	-0.3285
2009-01-09	Y	1.964
2009-01-10	Y	2.686
2009-01-01	Z	-1.779
2009-01-02	Z	2.399
2009-01-03	Z	-3.987
2009-01-04	Z	-2.831
2009-01-05	Z	1.437

Convert back to wide form

```
(api/pivot->wider stocks-long :stocks :price)
```

data/stockstidyr.csv [10 4]:

time	Z	X	Y
2009-01-01	-1.779	1.310	-1.890

time	Z	X	Y
2009-01-02	2.399	-0.2999	-1.825
2009-01-03	-3.987	0.5365	-1.036
2009-01-04	-2.831	-1.884	-0.5218
2009-01-05	1.437	-0.9605	-2.217
2009-01-06	3.398	-1.185	-2.894
2009-01-07	-1.201	-0.8521	-2.168
2009-01-08	-1.532	0.2523	-0.3285
2009-01-09	-6.809	0.4026	1.964
2009-01-10	-2.559	-0.6438	2.686

Convert to wide form on time column (let's limit values to a couple of rows)

```
(-> stocks-long
    (api/select-rows (range 0 30 4))
    (api/pivot->wider "time" :price))
```

data/stockstidyr.csv [3 6]:

:stocks	2009-01-05	2009-01-07	2009-01-01	2009-01-03	2009-01-09
X	-0.9605		1.310		0.4026
Z	1.437		-1.779		-6.809
Y		-2.168		-1.036	

Join/Concat

Left

Right

Inner

Hash

Concat

Functions