Proposed API for tech.ml.dataset

GenerateMe

2020-05-22

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	:mean	:mode	:min	:max	:standard-deviation	:skew
date	:packed-local-date	1461	0	2013-12-31		2012-01-01	2015-12-31		
precipitation	:float32	1461	0	3.029		0.000	55.90	6.680	3.506
temp_max	:float32	1461	0	16.44		-1.600	35.60	7.350	0.2809
temp_min	:float32	1461	0	8.235		-7.100	18.30	5.023	-0.2495
weather	:string	1461	0		sun				
wind	:float32	1461	0	3.241		0.4000	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 0x4fd2177e "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 0x3e89a0d6 "2012-01-01"],
  "precipitation" 0.0,
  "temp_min" 5.0,
```

```
  "weather" "drizzle",
  "temp_max" 12.8,
  "wind" 4.7}
 {"date" #object[java.time.LocalDate 0xd7dd8f1 "2012-01-02"],
  "precipitation" 10.9,
  "temp_min" 2.8,
  "weather" "rain",
  "temp_max" 10.6,
  "wind" 4.5})
```

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
[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 [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}))
```

(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 [4 4]:

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

:V1	:V2	:V3	:V4
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]}
```

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
```

(1 [2 2]:

a	b
1	a
1	b

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))
```

(1 [2 2]:

a	b
1	a
1	b

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 1 [2 2]:

a	b
1	a
1	b

, 2 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})
```

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

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

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

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

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

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

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

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

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

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

{: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-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-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})
```

(1.0 [2 4]:

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

2.0 [2 4]:

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

0.5 [2 4]:

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

3.0 [1 4]:

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

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})
```

(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

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})
```

([1 1.0] [1 4]:

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

[1 0.5] [2 4]:

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

[2 1.5] [1 4]:

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

[1 1.5] [2 4]:

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

[2 0.5] [1 4]:

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

[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]})
```

({: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

{: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 string" (str (:V4 row))}))
    (api/ungroup {:add-group-as-column true}))
```

_unnamed [9 6]:

V1 and V3 multiplied	V4 as string	:V1	:V2	:V3	:V4
3.000	C	2	6	1.500	C

V1 and V3 multiplied	V4 as string	:V1	:V2	:V3	:V4
1.500	C	1	3	1.500	C
1.500	C	1	9	1.500	C
1.000	A	2	4	0.5000	A
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

We can add group names without separation

```
(-> DS
    (api/group-by (fn [row] {"V1 and V3 multiplied" (* (:V1 row)
                                                       (:V3 row))
                             "V4 as string" (str (: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” 3.0, “V4 as string” “C”}	2	6	1.500	C
{“V1 and V3 multiplied” 1.5, “V4 as string” “C”}	1	3	1.500	C
{“V1 and V3 multiplied” 1.5, “V4 as string” “C”}	1	9	1.500	C
{“V1 and V3 multiplied” 1.0, “V4 as string” “A”}	2	4	0.5000	A
{“V1 and V3 multiplied” 0.5, “V4 as string” “A”}	1	1	0.5000	A
{“V1 and V3 multiplied” 0.5, “V4 as string” “A”}	1	7	0.5000	A
{“V1 and V3 multiplied” 1.0, “V4 as string” “B”}	1	5	1.000	B
{“V1 and V3 multiplied” 2.0, “V4 as string” “B”}	2	2	1.000	B
{“V1 and V3 multiplied” 2.0, “V4 as string” “B”}	2	8	1.000	B

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-filed`. `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 1 [5 2]:

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

, 2 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 1 [5 2]:

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

, 2 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@2b0acee5
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 1 [5 4]:

v1	v2	[1 2 3]	java.lang.Object@60b39df5
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 [4 4]:

v1	v2	[1 2 3]	java.lang.Object@60b39df5
2	2	1.000	B
2	4	0.5000	A
2	6	1.500	C
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
0.3120	1	0.5000	A
0.4625	2	1.000	B
0.4346	3	1.500	C
0.05148	4	0.5000	A
0.2961	5	1.000	B
0.1465	6	1.500	C
0.6781	7	0.5000	A
0.008779	8	1.000	B
0.4155	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

:V1	:V2	:V3	:V4	:row-count
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
3	6	1.500	C	:C	11
2	7	0.5000	A	:A	11
3	8	1.000	B	:B	11
2	9	1.500	C	:C	11

Map

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

Arguments:

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

Let's add numerical columns together

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

_unnamed [9 5]:

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

The same works on grouped dataset

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

_unnamed [9 5]:

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

Reorder

To reorder columns use columns selectors to choose what columns go first. The 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

---

Basic conversion

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

_unnamed :column info [4 5]:

:name	:size	:datatype	:unparsed-indexes	:unparsed-data
:V1	9	:float64	{}	[]
:V2	9	:int64		
:V3	9	:float64		
:V4	9	:object		

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 (str %) 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 :string})
    (api/info :columns))
```

_unnamed :column info [4 6]:

:name	:size	:datatype	:unparsed-indexes	:unparsed-data	:categorical?

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

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 5]:

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

Rows

Select

Drop

Other

Aggregate

Order

Unique

Strategies

Missing

Select

Drop

Replace

Join/Separate Columns

Join

Separate

Fold/Unroll Rows

Fold

Unroll

Reshape

Longer

Wider

Join/Concat

Left

Right

Inner

Hash

Concat

Functions