Dataset (data frame) manipulation API for the tech.ml.dataset library

GenerateMe

2022-06-20

```
tech-ml-version
```

"6.090"

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.

If you want to know more about `tech.ml.dataset` and `dtype-next` please refer their documentation:

- tech.ml.dataset walkthrough
- dtype-next overview
- dtype-next cheatsheet

SOURCE CODE

Join the discussion on Zulip

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

```
(require '[tablecloth.api :as tc]
         '[tech.v3.datatype.functional :as dfn])
(def DS (tc/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	0.5	A
2	2	1.0	B
1	3	1.5	C
2	4	0.5	A
1	5	1.0	B
2	6	1.5	C
1	7	0.5	A
2	8	1.0	B
1	9	1.5	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: `[string column-data]` or `[keyword column-data]`
- array of any arrays
- file types: raw/gzipped csv/tsv, json, xls(x) taken from local file system or URL
- input stream

`tc/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
  - `:column-names` - in case you want to name columns - only works for sequential input (arrays) or empty dataset
  - `:layout` - for numerical, native array of arrays - treat entries `:as-rows` or `:as-columns` (default)

`tc/let-dataset` accepts bindings `symbol-column-data` to simulate R's `tibble` function. Each binding is converted into a column. You can refer previous columns to in further bindings (as in `let`).

Empty dataset.

```
(tc/dataset)
```

```
_unnamed [0 0]
```

Empty dataset with column names

```
(tc/dataset nil {:column-names [:a :b]})
```

```
_unnamed [0 2]:

| :a | :b |
|----|----|
```

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

```
(tc/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.

```
(tc/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 names, vals = value(s)). Works the same as sequence of pairs.

```
(tc/dataset {:A 33})
(tc/dataset {:A [1 2 3]})
(tc/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

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

_unnamed [2 2]:

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

Sequence of maps

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

_unnamed [2 2]:

:b	:a
3	1
2	99

_unnamed [2 2]:

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

Missing values are marked by `nil`

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

_unnamed [3 2]:

:b	:a
1	
4	3
	11

Reading from arrays, by default `:as-rows`

```
(-> (map int-array [[1 2] [3 4] [5 6]])
    (into-array)
    (tc/dataset))
```

:_unnamed [3 2]:

0	1
1	2
3	4
5	6

`:as-columns`

```
(-> (map int-array [[1 2] [3 4] [5 6]])
    (into-array)
    (tc/dataset {:layout :as-columns}))
```

:_unnamed [2 3]:

0	1	2
1	3	5
2	4	6

`:as-rows` with names

```
(-> (map int-array [[1 2] [3 4] [5 6]])
    (into-array)
    (tc/dataset {:layout :as-rows
                 :column-names [:a :b]}))
```

:_unnamed [3 2]:

:a	:b
1	2
3	4
5	6

Any objects

```
(-> (map to-array [[:a :z] ["ee" "ww"] [9 10]])
    (into-array)
    (tc/dataset {:column-names [:a :b :c]
                 :layout :as-columns}))
```

:_unnamed [2 3]:

:a	:b	:c
:a	ee	9
:z	ww	10

Create dataset using macro `let-dataset` to simulate R `tibble` function. Each binding is converted into a column.

```
(tc/let-dataset [x (range 1 6)
                 y 1
                 z (dfn/+ x y)])
```

_unnamed [5 3]:

:x	:y	:z
1	1	2
2	1	3
3	1	4
4	1	5
5	1	6

Import CSV file

```
(tc/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 (tc/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	0.0	12.8	5.0	4.7	drizzle
2012-01-02	10.9	10.6	2.8	4.5	rain
2012-01-03	0.8	11.7	7.2	2.3	rain
2012-01-04	20.3	12.2	5.6	4.7	rain
2012-01-05	1.3	8.9	2.8	6.1	rain
2012-01-06	2.5	4.4	2.2	2.2	rain
2012-01-07	0.0	7.2	2.8	2.3	rain
2012-01-08	0.0	10.0	2.8	2.0	sun
2012-01-09	4.3	9.4	5.0	3.4	rain
2012-01-10	1.0	6.1	0.6	3.4	rain
...	...	...	...	...	...

date	precipitation	temp_max	temp_min	wind	weather
2015-12-21	27.4	5.6	2.8	4.3	rain
2015-12-22	4.6	7.8	2.8	5.0	rain
2015-12-23	6.1	5.0	2.8	7.6	rain
2015-12-24	2.5	5.6	2.2	4.3	rain
2015-12-25	5.8	5.0	2.2	1.5	rain
2015-12-26	0.0	4.4	0.0	2.5	sun
2015-12-27	8.6	4.4	1.7	2.9	rain
2015-12-28	1.5	5.0	1.7	1.3	rain
2015-12-29	0.0	7.2	0.6	2.6	fog
2015-12-30	0.0	5.6	-1.0	3.4	sun
2015-12-31	0.0	5.6	-2.1	3.5	sun

When none of above works, singleton dataset is created. Along with the error message from the exception thrown by `tech.ml.dataset`

```
(tc/dataset 999)
```

_unnamed [1 2]:

:*error*| :value	
Don't know how to create ISeq from: java.lang.Long	999

To see the stack trace, turn it on by setting `:stack-trace?` to `true`.

Set column name for single value. Also set the dataset name and turn off creating error message column.

```
(tc/dataset 999 {:single-value-column-name "my-single-value"
                 :error-column? false})
(tc/dataset 999 {:single-value-column-name ""
                 :dataset-name "Single value"
                 :error-column? false})
```

_unnamed [1 1]:

my-single-value
999

Single value [1 1]:

0
999

Saving Export dataset to a file or output stream can be done by calling `tc/write!`. 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.

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

```
nil
true
```

```
(tc/write! DS "output.nippy.gz")
```

Nippy

```
nil
```

```
(tc/dataset "output.nippy.gz")
```

output.nippy.gz [9 4]:

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

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

Number of rows

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

```
1461
```

Number of columns

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

```
6
```

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

```
(tc/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

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

https://vega.github.io/vega-lite/examples/data/seattle-weather.csv: descriptive-stats [6 12]:

:col-name	:datatype	:n-valid	:n-missing	:min	:mean	:mode	:max	:standard-deviation	:skew	:first	:last
date	:packed-local-date	1461	0	2012-01-01	2013-12-31		2015-12-31	3.64520463E+10	-3.63250384E-16	2012-01-01	2015-12-31
precipitation	:float64	1461	0	0.000	3.029		55.90	6.68019432E+00	3.05643721E+00	0.000	0.000
temp_max	:float64	1461	0	-1.600	16.44		35.60	7.34975810E+00	2.80929992E-01	12.80	5.600
temp_min	:float64	1461	0	-7.100	8.235		18.30	5.02300418E+00	-2.49458552E-01	5.000	-2.100
weather	:string	1461	0			rain				drizzle	sun
wind	:float64	1461	0	0.4000	3.241		9.500	1.43782506E+00	8.01667519E-01	4.700	3.500

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]:

:n-elems	:name	:categorical?	:datatype
1461	date		:packed-local-date
1461	precipitation		:float64
1461	temp_max		:float64
1461	temp_min		:float64
1461	wind		:float64
1461	weather	true	:string

Getting a dataset name

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

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

Setting a dataset name (operation is immutable).

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

Possible result types:

- `:as-seq` or `:as-seqs` - sequence of seqences (default)
- `:as-maps` - sequence of maps (rows)
- `:as-map` - map of sequences (columns)
- `:as-double-arrays` - array of double arrays

Select column.

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

```
#tech.v3.dataset.column<float64>[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.v3.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 (tc/columns ds))
```

```
(#tech.v3.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 (tc/columns ds :as-map))
```

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

Rows as sequence of sequences

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

```
([#object[java.time.LocalDate 0x1b1ddd68 "2012-01-01"] 0.0 12.8 5.0 4.7 "drizzle"] [#object[java.time.L
```

Select rows/columns as double-double-array

```
(-> ds
    (tc/select-columns :type/numerical)
    (tc/head)
    (tc/rows :as-double-arrays))
```

```
#object["[[D" 0x67511afe "[[D@67511afe"]
```

```
(-> ds
    (tc/select-columns :type/numerical)
    (tc/head)
    (tc/columns :as-double-arrays))
```

```
#object["[[D" 0x20111b69 "[[D@20111b69"]
```

Rows as sequence of maps

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

```
({"date" #object[java.time.LocalDate 0x34ba34fd "2012-01-01"],
  "precipitation" 0.0,
  "temp_min" 5.0,
  "weather" "drizzle",
  "temp_max" 12.8,
  "wind" 4.7}
 {"date" #object[java.time.LocalDate 0x420f047 "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`.

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

```
_unnamed [2 3]:

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

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

```
_unnamed [2 3]:

| :group-id | :name |                          :data |
|----------:|------:|--------------------------------|
|         0 |     1 | Group: 1 [5 4]:                |
|           |       |                                |
|           |       | \| :V1 \| :V2 \| :V3 \| :V4 \| |
|           |       | \|----:\|----:\|----:\|-----\| |
|           |       | \|   1 \|   1 \| 0.5 \|   A \| |
```

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

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

```
_unnamed [2 3]:

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

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
  - `:select-keys` - list of the columns passed to a grouping selector 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 (limited to `:select-keys`)

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 grouped dataset

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

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

List of columns in grouped dataset treated as regular dataset

```
(-> DS
    (tc/group-by :V1)
    (tc/as-regular-dataset)
    (tc/column-names))
```

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

Content of the grouped dataset

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

```
{:group-id #tech.v3.dataset.column<int64>[2]
:group-id
[0, 1], :name #tech.v3.dataset.column<int64>[2]
:name
[1, 2], :data #tech.v3.dataset.column<dataset>[2]
:data
[Group: 1 [5 4]:

| :V1 | :V2 | :V3 | :V4 |
|----:|----:|----:|-----|
|   1 |   1 | 0.5 |   A |
|   1 |   3 | 1.5 |   C |
|   1 |   5 | 1.0 |   B |
|   1 |   7 | 0.5 |   A |
|   1 |   9 | 1.5 |   C |
, Group: 2 [4 4]:

| :V1 | :V2 | :V3 | :V4 |
|----:|----:|----:|-----|
|   2 |   2 | 1.0 |   B |
|   2 |   4 | 0.5 |   A |
|   2 |   6 | 1.5 |   C |
|   2 |   8 | 1.0 |   B |
]}
```

Grouped dataset as map

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

```
(1 2)
```

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

(Group: 1 [5 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
1	3	1.5	C
1	5	1.0	B
1	7	0.5	A
1	9	1.5	C

Group: 2 [4 4]:

:V1	:V2	:V3	:V4
2	2	1.0	B
2	4	0.5	A
2	6	1.5	C
2	8	1.0	B

)

Group dataset as map of indexes (row ids)

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

```
{1 #list<int32>[5]
[0, 2, 4, 6, 8], 2 #list<int32>[4]
[1, 3, 5, 7]}
```

Grouped datasets are printed as follows by default.

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

_unnamed [2 3]:

:group-id	:name	:data
0	1	Group: 1 [5 4]:
1	2	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"]}
             (tc/dataset)
             (tc/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"]}
    (tc/dataset)
    (tc/group-by "a")
    (tc/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"]}
    (tc/dataset)
    (tc/group-by "a")
    (tc/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.

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

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

:V1	:V2	:V3	:V4
1	1	0.5	A
1	7	0.5	A

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

:V1	:V2	:V3	:V4
2	2	1.0	B
2	8	1.0	B

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

:V1	:V2	:V3	:V4
1	3	1.5	C
1	9	1.5	C

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

:V1	:V2	:V3	:V4
2	4	0.5	A

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

:V1	:V2	:V3	:V4
1	5	1.0	B

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

:V1	:V2	:V3	:V4
2	6	1.5	C

)

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

```
(tc/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	1.0	B
1	3	1.5	C
2	2	1.0	B
1	3	1.5	C

Group: group-b [4 4]:

:V1	:V2	:V3	:V4
2	6	1.5	C
2	6	1.5	C
2	6	1.5	C
2	2	1.0	B

)

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

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

(Group: 0.5 [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
1	7	0.5	A

Group: 2.0 [2 4]:

:V1	:V2	:V3	:V4
2	2	1.0	B
2	8	1.0	B

Group: 1.5 [2 4]:

:V1	:V2	:V3	:V4
1	3	1.5	C
1	9	1.5	C

Group: 1.0 [2 4]:

:V1	:V2	:V3	:V4
2	4	0.5	A
1	5	1.0	B

Group: 3.0 [1 4]:

:V1	:V2	:V3	:V4
2	6	1.5	C

)

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

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

(Group: true [3 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	4	0.5	A
1	7	0.5	A

Group: false [6 4]:

:V1	:V2	:V3	:V4
2	2	1.0	B
1	3	1.5	C
1	5	1.0	B
2	6	1.5	C
2	8	1.0	B

:V1	:V2	:V3	:V4
1	9	1.5	C

)

`juxt` is also helpful

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

(Group: [1 0.5] [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
1	7	0.5	A

Group: [2 1.0] [2 4]:

:V1	:V2	:V3	:V4
2	2	1.0	B
2	8	1.0	B

Group: [1 1.5] [2 4]:

:V1	:V2	:V3	:V4
1	3	1.5	C
1	9	1.5	C

Group: [2 0.5] [1 4]:

:V1	:V2	:V3	:V4
2	4	0.5	A

Group: [1 1.0] [1 4]:

:V1	:V2	:V3	:V4
1	5	1.0	B

Group: [2 1.5] [1 4]:

:V1	:V2	:V3	:V4
2	6	1.5	C

)

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

```
(tc/group-by DS identity {:result-type :as-seq
                          :select-keys [:V1]})
```

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

:V1	:V2	:V3	:V4
1	1	0.5	A
1	3	1.5	C
1	5	1.0	B
1	7	0.5	A
1	9	1.5	C

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

:V1	:V2	:V3	:V4
2	2	1.0	B
2	4	0.5	A
2	6	1.5	C
2	8	1.0	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
    (tc/group-by :V3)
    (tc/ungroup))
```

_unnamed [9 4]:

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

Groups sorted by group name and named.

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

Ordered by V3 [9 4]:

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

Groups sorted descending by group name and named.

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

Ordered by V3 descending [9 4]:

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

:V1	:V2	:V3	:V4
1	7	0.5	A

Let's add group name and id as additional columns

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

_unnamed [9 6]:

:*group* − *name*| :group-id	:V1	:V2	:V3	:V4	
true	0	1	1	0.5	A
true	0	2	2	1.0	B
true	0	1	3	1.5	C
false	1	2	4	0.5	A
false	1	1	5	1.0	B
false	1	2	6	1.5	C
false	1	1	7	0.5	A
false	1	2	8	1.0	B
false	1	1	9	1.5	C

Let's assign different column names

```
(-> DS
    (tc/group-by (comp #(< % 4) :V2))
    (tc/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
true	0	1	1	0.5	A
true	0	2	2	1.0	B
true	0	1	3	1.5	C
false	1	2	4	0.5	A
false	1	1	5	1.0	B
false	1	2	6	1.5	C
false	1	1	7	0.5	A
false	1	2	8	1.0	B
false	1	1	9	1.5	C

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

```
(-> DS
    (tc/group-by (fn [row] {"V1 and V3 multiplied" (* (:V1 row)
                                                      (:V3 row))
```

```
                            "V4 as lowercase" (clojure.string/lower-case (:V4 row))}))
    (tc/ungroup {:add-group-as-column true}))
```

_unnamed [9 6]:

V4 as lowercase	V1 and V3 multiplied	:V1	:V2	:V3	:V4
a	0.5	1	1	0.5	A
a	0.5	1	7	0.5	A
b	2.0	2	2	1.0	B
b	2.0	2	8	1.0	B
c	1.5	1	3	1.5	C
c	1.5	1	9	1.5	C
a	1.0	2	4	0.5	A
b	1.0	1	5	1.0	B
c	3.0	2	6	1.5	C

We can add group names without separation

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

_unnamed [9 5]:

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

The same applies to group names as sequences

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

_unnamed [9 6]:

:abc-0	:abc-1	:V1	:V2	:V3	:V4
1	0.5	1	1	0.5	A
1	0.5	1	7	0.5	A

:abc-0	:abc-1	:V1	:V2	:V3	:V4
2	1.0	2	2	1.0	B
2	1.0	2	8	1.0	B
1	1.5	1	3	1.5	C
1	1.5	1	9	1.5	C
2	0.5	2	4	0.5	A
1	1.0	1	5	1.0	B
2	1.5	2	6	1.5	C

Let's provide column names

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

_unnamed [9 6]:

v1	v3	:V1	:V2	:V3	:V4
1	0.5	1	1	0.5	A
1	0.5	1	7	0.5	A
2	1.0	2	2	1.0	B
2	1.0	2	8	1.0	B
1	1.5	1	3	1.5	C
1	1.5	1	9	1.5	C
2	0.5	2	4	0.5	A
1	1.0	1	5	1.0	B
2	1.5	2	6	1.5	C

Also we can supress separation

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

_unnamed [9 5]:

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

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

```
(tc/grouped? DS)
```

```
nil
```

```
(tc/grouped? (tc/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
    (tc/group-by :V1)
    (tc/as-regular-dataset)
    (tc/grouped?))
```

```
nil
```

You can also operate on grouped dataset as a regular one in case you want to access its columns using `without-grouping->` threading macro.

```
(-> DS
    (tc/group-by [:V4 :V1])
    (tc/without-grouping->
     (tc/order-by (comp (juxt :V4 :V1) :name))))
```

_unnamed [6 3]:

:group-id	:name	:data
0	{:V4 “A”, :V1 1}	Group: {:V4 “A”, :V1 1} [2 4]:
3	{:V4 “A”, :V1 2}	Group: {:V4 “A”, :V1 2} [1 4]:
4	{:V4 “B”, :V1 1}	Group: {:V4 “B”, :V1 1} [1 4]:
1	{:V4 “B”, :V1 2}	Group: {:V4 “B”, :V1 2} [2 4]:
2	{:V4 “C”, :V1 1}	Group: {:V4 “C”, :V1 1} [2 4]:
5	{:V4 “C”, :V1 2}	Group: {:V4 “C”, :V1 2} [1 4]:

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

_unnamed [2 3]:

:group-id	:name	:data
0	1	Shape: [5 4]
1	2	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
- `type` namespaced keyword for specific datatype or group of datatypes

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

Datatype groups are:

- `:type/numerical` - any numerical type
- `:type/float` - floating point number (`:float32` and `:float64`)
- `:type/integer` - any integer
- `:type/datetime` - any datetime type

If qualified keyword starts with `:!type`, complement set is used.

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

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

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

or

```
(tc/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.

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

```
()
```

Obviously selecting single name returns it's name if available

```
(tc/column-names DS :V1)
(tc/column-names DS "no such column")
```

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

Select sequence of column names.

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

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

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

```
(tc/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 `(tc/info DS :columns)`. Here we want to get integer columns only.

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

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

or

```
(tc/column-names DS :type/integer)
```

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

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

```
(tc/column-names DS #{:float64} :datatype)
```

```
(:V3)
```

or

```
(tc/column-names DS :type/float64)
```

```
(:V3)
```

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

```
(tc/column-names DS (complement #{:V1}))
(tc/column-names DS (complement #{:float64}) :datatype)
(tc/column-names DS :!type/float64)
```

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

```
(tc/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

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

_unnamed [9 1]:

:V3
0.5
1.0
1.5
0.5
1.0
1.5
0.5
1.0
1.5

or

```
(tc/select-columns DS :type/float64)
```

_unnamed [9 1]:

:V3
0.5
1.0
1.5
0.5
1.0
1.5
0.5
1.0
1.5

Select all but `:V1` columns

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

_unnamed [9 3]:

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

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

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

{1 Group: 1 [5 2]:

:V2	:V3
1	0.5
3	1.5
5	1.0
7	0.5
9	1.5

, 2 Group: 2 [4 2]:

:V2	:V3
2	1.0
4	0.5
6	1.5
8	1.0

}

Drop `drop-columns` creates dataset with removed columns.

Drop float64 columns

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

_unnamed [9 3]:

:V1	:V2	:V4
1	1	A
2	2	B

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

or

```
(tc/drop-columns DS :type/float64)
```

_unnamed [9 3]:

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

Drop all columns but `:V1` and `:V2`

```
(tc/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
    (tc/group-by :V1)
    (tc/drop-columns [:V2 :V3])
    (tc/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.

You can also pass mapping function with optional columns-selector

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

_unnamed [9 4]:

v1	v2	[1 2 3]	java.lang.Object@20f511c4
1	1	0.5	A
2	2	1.0	B
1	3	1.5	C
2	4	0.5	A
1	5	1.0	B
2	6	1.5	C
1	7	0.5	A
2	8	1.0	B
1	9	1.5	C

Map all names with function

```
(tc/rename-columns DS (comp str second name))
```

_unnamed [9 4]:

1	2	3	4
1	1	0.5	A
2	2	1.0	B
1	3	1.5	C
2	4	0.5	A
1	5	1.0	B
2	6	1.5	C
1	7	0.5	A
2	8	1.0	B
1	9	1.5	C

Map selected names with function

```
(tc/rename-columns DS [:V1 :V3] (comp str second name))
```

_unnamed [9 4]:

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

Function works on grouped dataset

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

{1 Group: 1 [5 4]:

v1	v2	[1 2 3]	java.lang.Object@175f5852
1	1	0.5	A
1	3	1.5	C
1	5	1.0	B
1	7	0.5	A
1	9	1.5	C

, 2 Group: 2 [4 4]:

v1	v2	[1 2 3]	java.lang.Object@175f5852
2	2	1.0	B
2	4	0.5	A
2	6	1.5	C
2	8	1.0	B

}

Add or update To add (or replace existing) column call `add-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` - repeat data
  - `:na` - append missing values
  - `:strict` - (default) throws an exception when sizes mismatch

Function works on grouped dataset.

Add single value as column

```
(tc/add-column DS :V5 "X")
```

_unnamed [9 5]:

:V1	:V2	:V3	:V4	:V5
1	1	0.5	A	X
2	2	1.0	B	X
1	3	1.5	C	X
2	4	0.5	A	X
1	5	1.0	B	X
2	6	1.5	C	X
1	7	0.5	A	X
2	8	1.0	B	X
1	9	1.5	C	X

Replace one column (column is trimmed)

```
(tc/add-column DS :V1 (repeatedly rand))
```

_unnamed [9 4]:

:V1	:V2	:V3	:V4
0.92974091	1	0.5	A
0.75657304	2	1.0	B
0.85343515	3	1.5	C

	:V1	:V2	:V3	:V4
0.80779884		4	0.5	A
0.57519958		5	1.0	B
0.09459056		6	1.5	C
0.90326301		7	0.5	A
0.94983248		8	1.0	B
0.07065385		9	1.5	C

Copy column

```
(tc/add-column DS :V5 (DS :V1))
```

_unnamed [9 5]:

:V1	:V2	:V3	:V4	:V5
1	1	0.5	A	1
2	2	1.0	B	2
1	3	1.5	C	1
2	4	0.5	A	2
1	5	1.0	B	1
2	6	1.5	C	2
1	7	0.5	A	1
2	8	1.0	B	2
1	9	1.5	C	1

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

```
(tc/add-column DS :row-count tc/row-count)
```

_unnamed [9 5]:

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

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

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

_unnamed [9 5]:

:V1	:V2	:V3	:V4	:row-count
1	1	0.5	A	5
1	3	1.5	C	5
1	5	1.0	B	5
1	7	0.5	A	5
1	9	1.5	C	5
2	2	1.0	B	4
2	4	0.5	A	4
2	6	1.5	C	4
2	8	1.0	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.

```
(tc/add-column DS :V5 [:r :b] :cycle)
```

_unnamed [9 5]:

:V1	:V2	:V3	:V4	:V5
1	1	0.5	A	:r
2	2	1.0	B	:b
1	3	1.5	C	:r
2	4	0.5	A	:b
1	5	1.0	B	:r
2	6	1.5	C	:b
1	7	0.5	A	:r
2	8	1.0	B	:b
1	9	1.5	C	:r

```
(tc/add-column DS :V5 [:r :b] :na)
```

_unnamed [9 5]:

:V1	:V2	:V3	:V4	:V5
1	1	0.5	A	:r
2	2	1.0	B	:b
1	3	1.5	C	
2	4	0.5	A	
1	5	1.0	B	
2	6	1.5	C	
1	7	0.5	A	
2	8	1.0	B	
1	9	1.5	C	

Exception is thrown when `:strict` (default) strategy is used and column size is not equal row count

```
(try
  (tc/add-column DS :V5 [:r :b])
  (catch Exception e (str "Exception caught: "(ex-message e))))
```

```
"Exception caught: Column size (2) should be exactly the same as dataset row count (9). Consider `:cycle
```

Tha same applies for grouped dataset

```
(-> DS
    (tc/group-by :V3)
    (tc/add-column :V5 [:r :b] :na)
    (tc/ungroup))
```

_unnamed [9 5]:

:V1	:V2	:V3	:V4	:V5
1	1	0.5	A	:r
2	4	0.5	A	:b
1	7	0.5	A	
2	2	1.0	B	:r
1	5	1.0	B	:b
2	8	1.0	B	
1	3	1.5	C	:r
2	6	1.5	C	:b
1	9	1.5	C	

Let's use other column to fill groups

```
(-> DS
    (tc/group-by :V3)
    (tc/add-column :V5 (DS :V2) :cycle)
    (tc/ungroup))
```

_unnamed [9 5]:

:V1	:V2	:V3	:V4	:V5
1	1	0.5	A	1
2	4	0.5	A	2
1	7	0.5	A	3
2	2	1.0	B	1
1	5	1.0	B	2
2	8	1.0	B	3
1	3	1.5	C	1
2	6	1.5	C	2
1	9	1.5	C	3

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

```
(tc/add-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	0.5	A	:A	11
3	2	1.0	B	:B	11
2	3	1.5	C	:C	11
3	4	0.5	A	:A	11
2	5	1.0	B	:B	11
3	6	1.5	C	:C	11
2	7	0.5	A	:A	11
3	8	1.0	B	:B	11
2	9	1.5	C	:C	11

Update If you want to modify specific column(s) you can call `update-columns`. Arguments:

- dataset
- one of:
  - `columns-selector` and function (or sequence of functions)
  - map where keys are column names and vals are function

Functions accept column and have to return column or sequence

---

Reverse of columns

```
(tc/update-columns DS :all reverse)
```

_unnamed [9 4]:

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

Apply dec/inc on numerical columns

```
(tc/update-columns DS :type/numerical [(partial map dec)
                                       (partial map inc)])
```

_unnamed [9 4]:

:V1	:V2	:V3	:V4
0	2	-0.5	A
1	3	0.0	B
0	4	0.5	C
1	5	-0.5	A
0	6	0.0	B

:V1	:V2	:V3	:V4
1	7	0.5	C
0	8	-0.5	A
1	9	0.0	B
0	10	0.5	C

You can also assign a function to a column by packing operations into the map.

```
(tc/update-columns DS {:V1 reverse
                       :V2 (comp shuffle seq)})
```

_unnamed [9 4]:

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

Map The other way of creating or updating column is to map rows 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
- `columns-selector` - columns selected
- `map-fn` - mapping function

Let's add numerical columns together

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

_unnamed [9 5]:

:V1	:V2	:V3	:V4	:sum-of-numbers
1	1	0.5	A	2.5
2	2	1.0	B	5.0
1	3	1.5	C	5.5
2	4	0.5	A	6.5
1	5	1.0	B	7.0
2	6	1.5	C	9.5

:V1	:V2	:V3	:V4	:sum-of-numbers
1	7	0.5	A	8.5
2	8	1.0	B	11.0
1	9	1.5	C	11.5

The same works on grouped dataset

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

_unnamed [9 5]:

:V1	:V2	:V3	:V4	:sum-of-numbers
1	1	0.5	A	2.5
2	4	0.5	A	6.5
1	7	0.5	A	8.5
2	2	1.0	B	5.0
1	5	1.0	B	7.0
2	8	1.0	B	11.0
1	3	1.5	C	5.5
2	6	1.5	C	9.5
1	9	1.5	C	11.5

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

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

_unnamed [9 4]:

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

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

_unnamed [9 4]:

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

Type conversion To convert column into given datatype can be done using `convert-types` 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
  - `column-selector` and `new-types` - column name and new datatype (or datatypes as sequence)

`new-types` can be:

- a type like `:int64` or `:string` or sequence of types
- or sequence of pair of datetype 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
    (tc/convert-types :V1 :float64)
    (tc/info :columns))
```

_unnamed :column info [4 6]:

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

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

```
(-> DS
    (tc/convert-types :V4 [[:int16 #(Integer/parseInt % 16)]]))
```

_unnamed [9 4]:

:V1	:V2	:V3	:V4
1	1	0.5	10
2	2	1.0	11
1	3	1.5	12
2	4	0.5	10
1	5	1.0	11
2	6	1.5	12
1	7	0.5	10
2	8	1.0	11
1	9	1.5	12

You can process several columns at once

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

_unnamed :column info [4 6]:

:n-elems	:name	:categorical?	:unparsed-indexes	:unparsed-data	:datatype
9	:V1		{}	[]	:float64
9	:V2		{}	[]	:int64
9	:V3		{}	[]	:boolean
9	:V4	true			:object

Convert one type into another

```
(-> DS
    (tc/convert-types :type/numerical :int16)
    (tc/info :columns))
```

_unnamed :column info [4 6]:

:n-elems	:name	:categorical?	:unparsed-indexes	:unparsed-data	:datatype
9	:V1		{}	[]	:int16
9	:V2		{}	[]	:int16
9	:V3		{}	[]	:int16
9	:V4	true			:string

Function works on the grouped dataset

```
(-> DS
    (tc/group-by :V1)
    (tc/convert-types :V1 :float32)
    (tc/ungroup)
    (tc/info :columns))
```

_unnamed :column info [4 6]:

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

Double array conversion.

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

```
#object["[J" 0x657b261f "[J@657b261f"]
```

Function also works on grouped dataset

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

```
(#object["[J" 0x2213e224 "[J@2213e224"] #object["[J" 0x364b8a5a "[J@364b8a5a"] #object["[J" 0x73e52b6e
```

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

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

```
#object["[Ljava.lang.String;" 0x252d8e57 "[Ljava.lang.String;@252d8e57"]
#object["[F" 0x438f83a8 "[F@438f83a8"]
```

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 (`select-keys` 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-columns` on a dataset. `:pre` is used as a column definitions.

Select Select fifth row

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

_unnamed [1 4]:

:V1	:V2	:V3	:V4
1	5	1.0	B

Select 3 rows

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

_unnamed [3 4]:

:V1	:V2	:V3	:V4
2	2	1.0	B
1	5	1.0	B
2	6	1.5	C

Select rows using sequence of true/false values

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

_unnamed [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	4	0.5	A

Select rows using predicate

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

_unnamed [3 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	4	0.5	A
1	7	0.5	A

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

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

_unnamed [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	2	1.0	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
    (tc/group-by :V4)
    (tc/select-rows (fn [row] (<= (:V2 row) (:mean row)))
                    {:pre {:mean #(tech.v3.datatype.functional/mean (% :V2))}})
    (tc/ungroup))
```

_unnamed [6 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	4	0.5	A
2	2	1.0	B
1	5	1.0	B
1	3	1.5	C
2	6	1.5	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
    (tc/group-by :V4)
    (tc/drop-rows (fn [row] (<= (:V2 row) (:mean row)))
                  {:pre {:mean #(tech.v3.datatype.functional/mean (% :V2))}})
    (tc/ungroup))
```

_unnamed [3 4]:

:V1	:V2	:V3	:V4
1	7	0.5	A
2	8	1.0	B
1	9	1.5	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.

All random functions accept `:seed` as an option if you want to fix returned result.

First row

```
(tc/first DS)
```

_unnamed [1 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A

Last row

```
(tc/last DS)
```

_unnamed [1 4]:

:V1	:V2	:V3	:V4
1	9	1.5	C

Random row (single)

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

_unnamed [1 4]:

:V1	:V2	:V3	:V4
2	2	1.0	B

Random row (single) with seed

```
(tc/rand-nth DS {:seed 42})
```

_unnamed [1 4]:

:V1	:V2	:V3	:V4
2	6	1.5	C

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

```
(tc/random DS)
```

_unnamed [9 4]:

:V1	:V2	:V3	:V4
2	2	1.0	B
2	4	0.5	A
1	1	0.5	A
2	2	1.0	B

:V1	:V2	:V3	:V4
1	1	0.5	A
1	1	0.5	A
1	3	1.5	C
2	4	0.5	A
1	5	1.0	B

Five random rows with repetition

```
(tc/random DS 5)
```

_unnamed [5 4]:

:V1	:V2	:V3	:V4
2	6	1.5	C
1	3	1.5	C
1	3	1.5	C
1	3	1.5	C
1	9	1.5	C

Five random, non-repeating rows

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

_unnamed [5 4]:

:V1	:V2	:V3	:V4
2	8	1.0	B
2	6	1.5	C
2	2	1.0	B
1	3	1.5	C
1	1	0.5	A

Five random, with seed

```
(tc/random DS 5 {:seed 42})
```

_unnamed [5 4]:

:V1	:V2	:V3	:V4
2	6	1.5	C
1	5	1.0	B
1	3	1.5	C
1	1	0.5	A
1	9	1.5	C

Shuffle dataset

```
(tc/shuffle DS)
```

_unnamed [9 4]:

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

Shuffle with seed

```
(tc/shuffle DS {:seed 42})
```

_unnamed [9 4]:

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

First `n` rows (default 5)

```
(tc/head DS)
```

_unnamed [5 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	2	1.0	B
1	3	1.5	C
2	4	0.5	A
1	5	1.0	B

Last `n` rows (default 5)

```
(tc/tail DS)
```

_unnamed [5 4]:

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

`by-rank` calculates rank on column(s). It's base on R rank() with addition of `:dense` (default) tie strategy which give consecutive rank numbering.

`:desc?` options (default: `true`) sorts input with descending order, giving top values under `0` value.

`rank` is zero based and is defined at `tablecloth.api.utils` namespace.

```
(tc/by-rank DS :V3 zero?) ;; most V3 values
```

_unnamed [3 4]:

:V1	:V2	:V3	:V4
1	3	1.5	C
2	6	1.5	C
1	9	1.5	C

```
(tc/by-rank DS :V3 zero? {:desc? false}) ;; least V3 values
```

_unnamed [3 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	4	0.5	A
1	7	0.5	A

Rank also works on multiple columns

```
(tc/by-rank DS [:V1 :V3] zero? {:desc? false})
```

_unnamed [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
1	7	0.5	A

Select 5 random rows from each group

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

_unnamed [15 4]:

:V1	:V2	:V3	:V4
1	7	0.5	A
1	7	0.5	A
1	7	0.5	A
1	7	0.5	A
1	1	0.5	A
2	8	1.0	B
2	8	1.0	B
2	8	1.0	B
2	2	1.0	B
1	5	1.0	B
2	6	1.5	C
1	9	1.5	C
1	3	1.5	C
1	9	1.5	C
2	6	1.5	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 off by setting `:ungroup` to false. In case you want to pass additional ungrouping parameters add them to the options.

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

Let's calculate mean of some columns

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

_unnamed [1 1]:

summary
45

Let's give resulting column a name.

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

_unnamed [1 1]:

:sum-of-V2
45

Sequential result is spread into separate columns

```
(tc/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

```
(tc/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	:V2-value-1-sum-v1	:V2-value-1-prod-v3
1	2	3	13	0.421875

Processing grouped dataset

```
(-> DS
    (tc/group-by [:V4])
    (tc/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]:

:V2-value-0-2	:V4	:V2-value-1-prod-v3	:V2-value-1-sum-v1	:V2-value-0-0	:V2-value-0-1
7	A	0.125	4	1	4
8	B	1.000	5	2	5
9	C	3.375	4	3	6

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

```
(-> DS
    (tc/group-by [:V3])
    (tc/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]:

:group-id	:name	:data
0	{:V3 0.5}	_unnamed [1 5]:
1	{:V3 1.0}	_unnamed [1 5]:
2	{:V3 1.5}	_unnamed [1 5]:

Column You can perform columnar aggreagation also. `aggregate-columns` selects columns and apply aggregating function (or sequence of functions) for each column separately.

```
(tc/aggregate-columns DS [:V1 :V2 :V3] #(reduce + %))
```

_unnamed [1 3]:

:V1	:V2	:V3
13	45	9.0

```
(tc/aggregate-columns DS [:V1 :V2 :V3] [#(reduce + %)
                                        #(reduce max %)
                                        #(reduce * %)])
```

_unnamed [1 3]:

:V1	:V2	:V3
13	9	0.421875

```
(-> DS
    (tc/group-by [:V4])
    (tc/aggregate-columns [:V1 :V2 :V3] #(reduce + %)))
```

_unnamed [3 4]:

:V2	:V4	:V3	:V1
12	A	1.5	4
15	B	3.0	5
18	C	4.5	4

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

`:select-keys` limits row map provided to ordering functions.

Order by single column, ascending

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

_unnamed [9 4]:

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

Descending order

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

_unnamed [9 4]:

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

Order by two columns

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

_unnamed [9 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
1	3	1.5	C
1	5	1.0	B

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

Use different orders for columns

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

_unnamed [9 4]:

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

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

_unnamed [9 4]:

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

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

_unnamed [9 4]:

:V1	:V2	:V3	:V4
2	4	0.5	A
2	2	1.0	B
2	8	1.0	B
2	6	1.5	C

:V1	:V2	:V3	:V4
1	1	0.5	A
1	7	0.5	A
1	5	1.0	B
1	3	1.5	C
1	9	1.5	C

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

```
(tc/order-by DS [:V4 (fn [row] (* (:V1 row)
                                  (:V2 row)
                                  (:V3 row)))] [:desc :asc])
```

_unnamed [9 4]:

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

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
```

```
(tc/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	0.5	A

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

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

_unnamed [9 4]:

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

Remove duplicates from each group selected by column.

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

_unnamed [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	2	1.0	B

Pair of columns

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

_unnamed [6 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	2	1.0	B
1	3	1.5	C
2	4	0.5	A
1	5	1.0	B
2	6	1.5	C

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

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

_unnamed [3 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	2	1.0	B
1	3	1.5	C

The same can be achived with `group-by`

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

_unnamed [3 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	2	1.0	B
1	3	1.5	C

Grouped dataset

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

_unnamed [6 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	4	0.5	A
2	2	1.0	B

:V1	:V2	:V3	:V4
1	5	1.0	B
1	3	1.5	C
2	6	1.5	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

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

_unnamed [2 4]:

:V1	:V2	:V3	:V4
2	8	1.0	B
1	9	1.5	C

Random

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

_unnamed [2 4]:

:V1	:V2	:V3	:V4
2	4	0.5	A
1	9	1.5	C

Pack columns into vector

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

_unnamed [3 3]:

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

Sum columns

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

_unnamed [3 3]:

:V2	:V3	:V1
12	1.5	4
15	3.0	5
18	4.5	4

Group by function and apply functions

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

_unnamed [3 4]:

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

Grouped dataset

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

_unnamed [6 5]:

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

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 (tc/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	0.5	A
2	2	1.0	B
	3		C
1	4	1.5	A
2	5	0.5	B
	6	1.0	C
1	7		A
2	8	1.5	B
	9	0.5	C

Select Select rows with missing values

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

_unnamed [4 4]:

:V1	:V2	:V3	:V4
	3		C
	6	1.0	C
1	7		A
	9	0.5	C

Select rows with missing values in `:V1`

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

_unnamed [3 4]:

:V1	:V2	:V3	:V4
	3		C
	6	1.0	C
	9	0.5	C

The same with grouped dataset

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

_unnamed [2 4]:

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

Drop Drop rows with missing values

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

_unnamed [5 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	2	1.0	B
1	4	1.5	A
2	5	0.5	B
2	8	1.5	B

Drop rows with missing values in `:V1`

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

_unnamed [6 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	2	1.0	B
1	4	1.5	A
2	5	0.5	B
1	7		A
2	8	1.5	B

The same with grouped dataset

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

_unnamed [6 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
1	4	1.5	A
1	7		A
2	2	1.0	B
2	5	0.5	B
2	8	1.5	B

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

- dataset
- column selector, default: `:all`
- strategy, default: `:nearest`
- value (optional)

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

Strategies are:

- `:value` - replace with given value
- `:up` - copy values up
- `:down` - copy values down
- `:updown` - copy values up and then down for missing values at the end
- `:downup` - copy values down and then up for missing values at the beginning
- `:mid` or `:nearest` - copy values around known values
- `:midpoint` - use average value from previous and next non-missing
- `:lerp` - trying to lineary approximate values, works for numbers and datetime, otherwise applies `:nearest`. For numbers always results in `float` datatype.

Let's define special dataset here:

```
(def DSm2 (tc/dataset {:a [nil nil nil 1.0 2  nil nil nil nil  nil 4   nil  11 nil nil]
                       :b [2   2   2 nil nil nil nil nil nil 13   nil   3  4  5 5]}))
```

```
DSm2
```

_unnamed [15 2]:

:a	:b
	2
	2
	2
1.0	
2.0	
	13
4.0	
	3
11.0	4
	5
	5

Replace missing with default strategy for all columns

```
(tc/replace-missing DSm2)
```

_unnamed [15 2]:

:a	:b
1.0	2
1.0	2
1.0	2
1.0	2
2.0	2

:a	:b
2.0	2
2.0	13
2.0	13
4.0	13
4.0	13
4.0	13
4.0	3
11.0	4
11.0	5
11.0	5

Replace missing with single value in whole dataset

```
(tc/replace-missing DSm2 :all :value 999)
```

_unnamed [15 2]:

:a	:b
999.0	2
999.0	2
999.0	2
1.0	999
2.0	999
999.0	999
999.0	999
999.0	999
999.0	999
999.0	13
4.0	999
999.0	3
11.0	4
999.0	5
999.0	5

Replace missing with single value in `:a` column

```
(tc/replace-missing DSm2 :a :value 999)
```

_unnamed [15 2]:

:a	:b
999.0	2
999.0	2
999.0	2
1.0	
2.0	
999.0	
999.0	

:a	:b
999.0	
999.0	
999.0	13
4.0	
999.0	3
11.0	4
999.0	5
999.0	5

Replace missing with sequence in `:a` column

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

_unnamed [15 2]:

:a	:b
-999.0	2
-998.0	2
-997.0	2
1.0	
2.0	
-999.0	
-998.0	
-997.0	
-999.0	
-998.0	13
4.0	
-997.0	3
11.0	4
-999.0	5
-998.0	5

Replace missing with a function (mean)

```
(tc/replace-missing DSm2 :a :value tech.v3.datatype.functional/mean)
```

_unnamed [15 2]:

:a	:b
4.5	2
4.5	2
4.5	2
1.0	
2.0	
4.5	
4.5	
4.5	
4.5	

:a	:b
4.5	13
4.0	
4.5	3
11.0	4
4.5	5
4.5	5

Using `:down` strategy, fills gaps with values from above. You can see that if missings are at the beginning, the are filled with first value

```
(tc/replace-missing DSm2 [:a :b] :downup)
```

_unnamed [15 2]:

:a	:b
1.0	2
1.0	2
1.0	2
1.0	2
2.0	2
2.0	2
2.0	2
2.0	2
2.0	2
2.0	13
4.0	13
4.0	3
11.0	4
11.0	5
11.0	5

To fix above issue you can provide value

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

_unnamed [15 2]:

:a	:b
999.0	2
999.0	2
999.0	2
1.0	2
2.0	2
2.0	2
2.0	2
2.0	2
2.0	2
2.0	13

:a	:b
4.0	13
4.0	3
11.0	4
11.0	5
11.0	5

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

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

_unnamed [15 2]:

:a	:b
1.0	2
1.0	2
1.0	2
1.0	13
2.0	13
4.0	13
4.0	13
4.0	13
4.0	13
4.0	13
4.0	3
11.0	3
11.0	4
	5
	5

```
(tc/replace-missing DSm2 [:a :b] :updown)
```

_unnamed [15 2]:

:a	:b
1.0	2
1.0	2
1.0	2
1.0	13
2.0	13
4.0	13
4.0	13
4.0	13
4.0	13
4.0	13
4.0	3
11.0	3
11.0	4
11.0	5

:a	:b
11.0	5

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

```
(tc/replace-missing DSm2 [:a :b] :midpoint)
```

_unnamed [15 2]:

:a	:b
1.0	2.0
1.0	2.0
1.0	2.0
1.0	7.5
2.0	7.5
3.0	7.5
3.0	7.5
3.0	7.5
3.0	7.5
3.0	13.0
4.0	8.0
7.5	3.0
11.0	4.0
11.0	5.0
11.0	5.0

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

```
(tc/replace-missing DSm2 [:a :b] :down tech.v3.datatype.functional/mean)
```

_unnamed [15 2]:

:a	:b
4.5	2
4.5	2
4.5	2
1.0	2
2.0	2
2.0	2
2.0	2
2.0	2
2.0	2
2.0	13
4.0	13
4.0	3
11.0	4
11.0	5
11.0	5

Lerp tries to apply linear interpolation of the values

```
(tc/replace-missing DSm2 [:a :b] :lerp)
```

_unnamed [15 2]:

:a	:b
1.00000000	2.00000000
1.00000000	2.00000000
1.00000000	2.00000000
1.00000000	3.57142857
2.00000000	5.14285714
2.33333333	6.71428571
2.66666667	8.28571429
3.00000000	9.85714286
3.33333333	11.42857143
3.66666667	13.00000000
4.00000000	8.00000000
7.50000000	3.00000000
11.00000000	4.00000000
11.00000000	5.00000000
11.00000000	5.00000000

Lerp works also on dates

```
(-> (tc/dataset {:dt [(java.time.LocalDateTime/of 2020 1 1 11 22 33)
                      nil nil nil nil nil nil nil
                      (java.time.LocalDateTime/of 2020 10 1 1 1 1)]})
    (tc/replace-missing :lerp))
```

_unnamed [9 1]:

:dt
2020-01-01T11:22:33
2020-02-04T16:04:51.500
2020-03-09T20:47:10
2020-04-13T01:29:28.500
2020-05-17T06:11:47
2020-06-20T10:54:05.500
2020-07-24T15:36:24
2020-08-27T20:18:42.500
2020-10-01T01:01:01

Inject When your column contains not continuous data range you can fill up with lacking values. Arguments:

- dataset
- column name
- expected step (`max-span`, milliseconds in case of datetime column)
- (optional) `missing-strategy` - how to replace missing, default `:down` (set to `nil` if none)
- (optional) `missing-value` - optional value for replace missing

```
(-> (tc/dataset {:a [1 2 9]
                 :b [:a :b :c]})
    (tc/fill-range-replace :a 1))
```

_unnamed [9 2]:

:a	:b
1.0	:a
2.0	:b
3.0	:b
4.0	:b
5.0	:b
6.0	:b
7.0	:b
8.0	:b
9.0	:c

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.

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

_unnamed [9 2]:

:V3	:joined
0.5	1-1-A
1.0	2-2-B
	3-C
1.5	1-4-A
0.5	2-5-B
1.0	6-C

:V3	:joined
	1-7-A
1.5	2-8-B
0.5	9-C

Without dropping source columns.

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

_unnamed [9 5]:

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

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

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

_unnamed [9 2]:

:V3	:joined
0.5	1-1-A
1.0	2-2-B
	NA-3-C
1.5	1-4-A
0.5	2-5-B
1.0	NA-6-C
	1-7-A
1.5	2-8-B
0.5	NA-9-C

We can use custom separator.

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

_unnamed [9 2]:

:V3	:joined
0.5	1/1/A
1.0	2/2/B
	./3/C
1.5	1/4/A
0.5	2/5/B
1.0	./6/C
	1/7/A
1.5	2/8/B
0.5	./9/C

Or even sequence of separators.

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

_unnamed [9 2]:

:V3	:joined
0.5	1-1/A
1.0	2-2/B
	.-3/C
1.5	1-4/A
0.5	2-5/B
1.0	.-6/C
	1-7/A
1.5	2-8/B
0.5	.-9/C

The other types of results, map:

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

_unnamed [9 2]:

:V3	:joined
0.5	{:V1 1, :V2 1, :V4 “A”}
1.0	{:V1 2, :V2 2, :V4 “B”}
	{:V1 nil, :V2 3, :V4 “C”}
1.5	{:V1 1, :V2 4, :V4 “A”}
0.5	{:V1 2, :V2 5, :V4 “B”}
1.0	{:V1 nil, :V2 6, :V4 “C”}
	{:V1 1, :V2 7, :V4 “A”}
1.5	{:V1 2, :V2 8, :V4 “B”}
0.5	{:V1 nil, :V2 9, :V4 “C”}

Sequence

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

_unnamed [9 2]:

:V3	:joined
0.5	(1 1 “A”)
1.0	(2 2 “B”)
	(nil 3 “C”)
1.5	(1 4 “A”)
0.5	(2 5 “B”)
1.0	(nil 6 “C”)
	(1 7 “A”)
1.5	(2 8 “B”)
0.5	(nil 9 “C”)

Custom function, calculate hash

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

_unnamed [9 2]:

:V3	:joined
0.5	535226087
1.0	1128801549
	-1842240303
1.5	2022347171
0.5	1884312041
1.0	-1555412370
	1640237355
1.5	-967279152
0.5	1128367958

Grouped dataset

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

_unnamed [9 2]:

:V3	:joined
0.5	1-1-A
1.5	1-4-A
	1-7-A
1.0	2-2-B
0.5	2-5-B
1.5	2-8-B
	3-C

:V3	:joined
1.0	6-C
0.5	9-C

Tidyr examples source

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

```
#'user/df
```

```
df
```

_unnamed [4 2]:

:x	:y
a	b
a	
	b

```
(tc/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

```
(tc/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 - can be `nil` or `:infer` if `separator` returns map
- 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(s) or not (default: `true` or `:all` in case of empty `target-columns`). When set to `:all` keeps only separation result.
  - `: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

```
(tc/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	0.5	A
2	2	1	0.0	B
1	3	1	0.5	C
2	4	0	0.5	A
1	5	1	0.0	B
2	6	1	0.5	C
1	7	0	0.5	A
2	8	1	0.0	B
1	9	1	0.5	C

Source column can be kept

```
(tc/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	0.5	0	0.5	A
2	2	1.0	1	0.0	B
1	3	1.5	1	0.5	C
2	4	0.5	0	0.5	A
1	5	1.0	1	0.0	B
2	6	1.5	1	0.5	C

:V1	:V2	:V3	:int-part	:frac-part	:V4
1	7	0.5	0	0.5	A
2	8	1.0	1	0.0	B
1	9	1.5	1	0.5	C

We can treat `0` or `0.0` as missing value

```
(tc/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		0.5	A
2	2	1		B
1	3	1	0.5	C
2	4		0.5	A
1	5	1		B
2	6	1	0.5	C
1	7		0.5	A
2	8	1		B
1	9	1	0.5	C

Works on grouped dataset

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

_unnamed [9 5]:

:V1	:V2	:int-part	:fract-part	:V4
1	1	0	0.5	A
2	4	0	0.5	A
1	7	0	0.5	A
2	2	1	0.0	B
1	5	1	0.0	B
2	8	1	0.0	B
1	3	1	0.5	C
2	6	1	0.5	C
1	9	1	0.5	C

Separate using separator returning sequence of maps, in this case we drop all other columns.

```
(tc/separate-column DS :V3 (fn [^double v]
                             {:int-part (int (quot v 1.0))
                              :fract-part (mod v 1.0)}))
```

_unnamed [9 2]:

:int-part	:fract-part
0	0.5
1	0.0
1	0.5
0	0.5
1	0.0
1	0.5
0	0.5
1	0.0
1	0.5

Keeping all columns

```
(tc/separate-column DS :V3 nil (fn [^double v]
                                 {:int-part (int (quot v 1.0))
                                  :fract-part (mod v 1.0)}) {:drop-column? false})
```

_unnamed [9 6]:

:V1	:V2	:V3	:int-part	:fract-part	:V4
1	1	0.5	0	0.5	A
2	2	1.0	1	0.0	B
1	3	1.5	1	0.5	C
2	4	0.5	0	0.5	A
1	5	1.0	1	0.0	B
2	6	1.5	1	0.5	C
1	7	0.5	0	0.5	A
2	8	1.0	1	0.0	B
1	9	1.5	1	0.5	C

Join and separate together.

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

_unnamed [9 4]:

:V3	:v1	:v2	:v4
0.5	1	1	A
1.0	2	2	B
		3	C
1.5	1	4	A
0.5	2	5	B

:V3	:v1	:v2	:v4
1.0		6	C
	1	7	A
1.5	2	8	B
0.5		9	C

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

_unnamed [9 4]:

:V3	:v1	:v2	:v4
0.5	1	1	A
1.0	2	2	B
		3	C
1.5	1	4	A
0.5	2	5	B
1.0		6	C
	1	7	A
1.5	2	8	B
0.5		9	C

Tidyr examples separate source extract source

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

```
(tc/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.

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

_unnamed [4 1]:

:B
b
d

:B
c

Extra data is dropped

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

_unnamed [4 2]:

a	b
a	
a	b
a	b

Split with regular expression

```
(tc/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

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

_unnamed [4 2]:

a	b
a	b
a	b
b	c

Extract first value only

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

_unnamed [5 1]:

A
a
a
b
d

Split with regex

```
(tc/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

```
(tc/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

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

_unnamed [6 4]:

:V4	:V3	:V1	:V2
A	0.5	1	[1 7]
B	1.0	2	[2 8]

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

You can pack several columns at once.

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

_unnamed [3 4]:

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

You can use custom packing function

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

_unnamed [3 4]:

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

or

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

_unnamed [3 4]:

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

This works also on grouped dataset

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

_unnamed [6 4]:

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

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

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

_unnamed [9 4]:

:V4	:V2	:V3	:V1
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
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

Unroll all folded columns

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

_unnamed [9 4]:

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

:V4	:V1	:V2	:V3
C	1	9	1.5

Unroll one by one leads to cartesian product

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

_unnamed [15 4]:

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

You can add indexes

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

_unnamed [9 5]:

:V1	:indexes	:V4	:V2	:V3
1	0	A	1	0.5
1	1	C	3	1.5
1	2	B	5	1.0
1	3	A	7	0.5
1	4	C	9	1.5
2	0	B	2	1.0
2	1	A	4	0.5
2	2	C	6	1.5
2	3	B	8	1.0

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

_unnamed [9 5]:

:V1	vector idx	:V4	:V2	:V3
1	0	A	1	0.5
1	1	C	3	1.5
1	2	B	5	1.0
1	3	A	7	0.5
1	4	C	9	1.5
2	0	B	2	1.0
2	1	A	4	0.5
2	2	C	6	1.5
2	3	B	8	1.0

You can also force datatypes

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

_unnamed :column info [4 4]:

:n-elems	:name	:categorical?	:datatype
9	:V1		:int64
9	:V4	true	:string
9	:V2		:int16
9	:V3		:float32

This works also on grouped dataset

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

_unnamed [9 5]:

:V4	:V1	:V2	:indexes	:V3
A	1	[1 7]	0	0.5
A	1	[1 7]	1	0.5
C	1	[3 9]	0	1.5
C	1	[3 9]	1	1.5
B	1	[5]	0	1.0
B	2	[2 8]	0	1.0
B	2	[2 8]	1	1.0
A	2	[4]	0	0.5
C	2	[6]	0	1.5

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` - names of the columns created or columns pattern (see below) (default: `:$column`)
  - `:value-column-name` - name of the column for values (default: `:$value`)
  - `:splitter` - string, 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 (tc/dataset "data/relig_income.csv"))
```

```
relig-income
```

data/relig_income.csv [18 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
Evangelical Prot	575	869	1064	982	881	1486	949	723	414	1529
Hindu	1	9	7	9	11	34	47	48	54	37
Historically Black Prot	228	244	236	238	197	223	131	81	78	339
Jehovah's Witness	20	27	24	24	21	30	15	11	6	37

religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k	$50-75k	$75-100k	$100-150k	>150k	Don't know/refused
Jewish	19	19	25	25	30	95	69	87	151	162
Mainline Prot	289	495	619	655	651	1107	939	753	634	1328
Mormon	29	40	48	51	56	112	85	49	42	69
Muslim	6	7	9	10	9	23	16	8	6	22
Orthodox	13	17	23	32	32	47	38	42	46	73
Other Christian	9	7	11	13	13	14	18	14	12	18
Other Faiths	20	33	40	46	49	63	46	40	41	71
Other World Religions	5	2	3	4	2	7	3	4	4	8
Unaffiliated	217	299	374	365	341	528	407	321	258	597

```
(tc/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
...	...	...
Historically Black Prot	>150k	78
Jehovah's Witness	>150k	6
Jewish	>150k	151
Mainline Prot	>150k	634
Mormon	>150k	42
Muslim	>150k	6
Orthodox	>150k	46
Other Christian	>150k	12
Other Faiths	>150k	41
Other World Religions	>150k	4
Unaffiliated	>150k	258

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

```
(def bilboard (-> (tc/dataset "data/billboard.csv.gz")
                  (tc/drop-columns :type/boolean))) ;; drop some boolean columns, tidyr just skips them
```

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

data/billboard.csv.gz [317 13]:

artist	track	date.entered	wk1	wk2	wk3	wk4	wk5	wk6	wk7	wk8	wk9
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
3 Doors Down	Loser	2000-10-21	76	76	72	69	67	65	55	59	62
504 Boyz	Wobble Wobble	2000-04-15	57	34	25	17	17	31	36	49	53
98ˆ0	Give Me Just One Nig...	2000-08-19	51	39	34	26	26	19	2	2	3
A*Teens	Dancing Queen	2000-07-08	97	97	96	95	100				
Aaliyah	I Don't Wanna	2000-01-29	84	62	51	41	38	35	35	38	38
Aaliyah	Try Again	2000-03-18	59	53	38	28	21	18	16	14	12
Adams, Yolanda	Open My Heart	2000-08-26	76	76	74	69	68	67	61	58	57
...	...	...	...	...	...	...	...	...	...	...	...
Wallflowers, The	Sleepwalker	2000-10-28	73	73	74	80	90	96			
Westlife	Swear It Again	2000-04-01	96	82	66	55	55	46	44	44	37
Williams, Robbie	Angels	1999-11-20	85	77	69	69	62	56	56	64	54
Wills, Mark	Back At One	2000-01-15	89	55	51	43	37	37	36	39	42
Worley, Darryl	When You Need My Lov...	2000-06-17	98	88	93	92	85	85	84	80	80
Wright, Chely	It Was	2000-03-04	86	78	75	72	71	69	64	75	85
Yankee Grey	Another Nine Minutes	2000-04-29	86	83	77	74	83	79	88	95	
Yearwood, Trisha	Real Live Woman	2000-04-01	85	83	83	82	81	91			
Ying Yang Twins	Whistle While You Tw...	2000-03-18	95	94	91	85	84	78	74	78	85
Zombie Nation	Kernkraft 400	2000-09-02	99	99							
matchbox twenty	Bent	2000-04-29	60	37	29	24	22	21	18	16	13

```
(tc/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
matchbox twenty	Bent	2000-04-29	wk35	33
Creed	Higher	1999-09-11	wk55	21
...	...	...	...	...
Savage Garden	I Knew I Loved You	1999-10-23	wk24	12
Sisqo	Incomplete	2000-06-24	wk24	31
Sisqo	Thong Song	2000-01-29	wk24	17
Smash Mouth	Then The Morning Com...	1999-10-30	wk24	35
Son By Four	A Puro Dolor (Purest...	2000-04-08	wk24	32
Sonique	It Feels So Good	2000-01-22	wk24	49
SoulDecision	Faded	2000-07-08	wk24	50
Sting	Desert Rose	2000-05-13	wk24	45
Train	Meet Virginia	1999-10-09	wk24	42

artist	track	date.entered	:week	:rank
Vertical Horizon	Everything You Want	2000-01-22	wk24	6
matchbox twenty	Bent	2000-04-29	wk24	9

We can create numerical column out of column names

```
(tc/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
...	...	...	...	...
matchbox twenty	Bent	2000-04-29	5	22
3 Doors Down	Kryptonite	2000-04-08	34	3
Braxton, Toni	He Wasn't Man Enough	2000-03-18	34	33
Creed	Higher	1999-09-11	34	23
Creed	With Arms Wide Open	2000-05-13	34	5
Hill, Faith	Breathe	1999-11-06	34	5
Joe	I Wanna Know	2000-01-01	34	8
Lonestar	Amazed	1999-06-05	34	17
Nelly	(Hot S**t) Country G...	2000-04-29	34	49
Vertical Horizon	Everything You Want	2000-01-22	34	20
matchbox twenty	Bent	2000-04-29	34	30

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

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

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

data/who.csv.gz [7240 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						
Afghanistan	AF	AFG	1985						
Afghanistan	AF	AFG	1986						
Afghanistan	AF	AFG	1987						
Afghanistan	AF	AFG	1988						
Afghanistan	AF	AFG	1989						
...	...	...	...	...	...	...	...	...	...
Zimbabwe	ZW	ZWE	2003	133	874	3048	2228	981	367
Zimbabwe	ZW	ZWE	2004	187	833	2908	2298	1056	366
Zimbabwe	ZW	ZWE	2005	210	837	2264	1855	762	295
Zimbabwe	ZW	ZWE	2006	215	736	2391	1939	896	348
Zimbabwe	ZW	ZWE	2007	138	500	3693	0	716	292
Zimbabwe	ZW	ZWE	2008	127	614	0	3316	704	263
Zimbabwe	ZW	ZWE	2009	125	578		3471	681	293
Zimbabwe	ZW	ZWE	2010	150	710	2208	1682	761	350
Zimbabwe	ZW	ZWE	2011	152	784	2467	2071	780	377
Zimbabwe	ZW	ZWE	2012	120	783	2421	2086	796	360
Zimbabwe	ZW	ZWE	2013						

```
(tc/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
...	...	...	...	...	...	...	...
United Arab Emirates	AE	ARE	2013	rel	m	2534	9
United Kingdom of Great Britain and Northern Ireland	GB	GBR	2013	rel	m	2534	1158
United States of America	US	USA	2013	rel	m	2534	829
Uruguay	UY	URY	2013	rel	m	2534	142
Uzbekistan	UZ	UZB	2013	rel	m	2534	2371
Vanuatu	VU	VUT	2013	rel	m	2534	9
Venezuela (Bolivarian Republic of)	VE	VEN	2013	rel	m	2534	739
Viet Nam	VN	VNM	2013	rel	m	2534	6302
Yemen	YE	YEM	2013	rel	m	2534	1113

country	iso2	iso3	year	:diagnosis	:gender	:age	:count
Zambia	ZM	ZMB	2013	rel	m	2534	7808
Zimbabwe	ZW	ZWE	2013	rel	m	2534	5331

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 (tc/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

```
(tc/pivot->longer family (complement #{"family"}) {:target-columns [nil :child]
                                                   :splitter "_"
                                                   :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 (tc/dataset "data/anscombe.csv"))
```

```
anscombe
```

data/anscombe.csv [11 8]:

x1	x2	x3	x4	y1	y2	y3	y4
10	10	10	8	8.04	9.14	7.46	6.58
8	8	8	8	6.95	8.14	6.77	5.76

x1	x2	x3	x4	y1	y2	y3	y4
13	13	13	8	7.58	8.74	12.74	7.71
9	9	9	8	8.81	8.77	7.11	8.84
11	11	11	8	8.33	9.26	7.81	8.47
14	14	14	8	9.96	8.10	8.84	7.04
6	6	6	8	7.24	6.13	6.08	5.25
4	4	4	19	4.26	3.10	5.39	12.50
12	12	12	8	10.84	9.13	8.15	5.56
7	7	7	8	4.82	7.26	6.42	7.91
5	5	5	8	5.68	4.74	5.73	6.89

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

data/anscombe.csv [44 3]:

:set	x	y
1	10	8.04
1	8	6.95
1	13	7.58
1	9	8.81
1	11	8.33
1	14	9.96
1	6	7.24
1	4	4.26
1	12	10.84
1	7	4.82
...	...	...
4	8	6.58
4	8	5.76
4	8	7.71
4	8	8.84
4	8	8.47
4	8	7.04
4	8	5.25
4	19	12.50
4	8	5.56
4	8	7.91
4	8	6.89

```
(def pnl (tc/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	0.43183480	0.51173598	3	-2
2	1	1	0.86968631	0.87719400	3	-2
3	0	1	0.39852736	0.06539204	3	-2
4	0	1	0.71392261	0.25437648	3	-2

```
(tc/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	0.43183480	3
2	1	1	1	0.86968631	3
3	0	1	1	0.39852736	3
4	0	1	1	0.71392261	3
1	1	0	2	0.51173598	-2
2	1	1	2	0.87719400	-2
3	0	1	2	0.06539204	-2
4	0	1	2	0.25437648	-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 `:concat-columns-with` option. `:concat-columns-with` can be a string or function (default: `"_"`). Function accepts sequence of names.

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

When `value-columns` is a sequence, multiple observations as columns are created appending value column names into new columns. Column names are joined using `:concat-value-with` option. `:concat-value-with` can be a string or function (default: "-"). Function accepts current column name and value.

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

```
(def fish (tc/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

fish	station	seen
4842	BCE	1
4842	BCW	1
4842	BCE2	1
4842	BCW2	1
4842	MAE	1
...	...	...
4862	BCE	1
4862	BCW	1
4862	BCE2	1
4862	BCW2	1
4863	Release	1
4863	I80_1	1
4864	Release	1
4864	I80_1	1
4865	Release	1
4865	I80_1	1
4865	Lisbon	1

```
(tc/pivot->wider fish "station" "seen" {:drop-missing? false})
```

data/fish_encounters.csv [19 12]:

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

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

```
(def warpbreaks (tc/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
...	...	...
39	B	M
29	B	M
20	B	H
21	B	H
24	B	H
17	B	H
13	B	H
15	B	H
15	B	H
16	B	H
28	B	H

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

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

_unnamed [6 3]:

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

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

data/warpbreaks.csv [3 3]:

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

We can also calculate mean (aggreate values)

```
(-> warpbreaks
    (tc/reorder-columns ["wool" "tension" "breaks"])
    (tc/pivot->wider "wool" "breaks" {:fold-fn tech.v3.datatype.functional/mean}))
```

data/warpbreaks.csv [3 3]:

A	tension	B
44.55555556	L	28.22222222
24.00000000	M	28.77777778
24.55555556	H	18.77777778

Multiple source columns, joined with default separator.

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

```
production
```

data/production.csv [45 4]:

product	country	year	production
A	AI	2000	1.63727158
A	AI	2001	0.15870784
A	AI	2002	-1.56797745
A	AI	2003	-0.44455509
A	AI	2004	-0.07133701
A	AI	2005	1.61183090
A	AI	2006	-0.70434682
A	AI	2007	-1.53550542
A	AI	2008	0.83907155
A	AI	2009	-0.37424110
...	...	...	...
B	EI	2004	0.62564999
B	EI	2005	-1.34530299
B	EI	2006	-0.97184975
B	EI	2007	-1.69715821
B	EI	2008	0.04556128
B	EI	2009	1.19315043
B	EI	2010	-1.60557503
B	EI	2011	-0.77235497
B	EI	2012	-2.50262738
B	EI	2013	-1.62753769
B	EI	2014	0.03329645

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

data/production.csv [15 4]:

year	A_AI	B_AI	B_EI
2000	1.63727158	-0.02617661	1.40470848

year	A_AI	B_AI	B_EI
2001	0.15870784	-0.68863576	-0.59618369
2002	-1.56797745	0.06248741	-0.26568579
2003	-0.44455509	-0.72339686	0.65257808
2004	-0.07133701	0.47248952	0.62564999
2005	1.61183090	-0.94173861	-1.34530299
2006	-0.70434682	-0.34782108	-0.97184975
2007	-1.53550542	0.52425284	-1.69715821
2008	0.83907155	1.83230937	0.04556128
2009	-0.37424110	0.10706491	1.19315043
2010	-0.71158926	-0.32903664	-1.60557503
2011	1.12805634	-1.78319121	-0.77235497
2012	1.45718247	0.61125798	-2.50262738
2013	-1.55934101	-0.78526092	-1.62753769
2014	-0.11695838	0.97843635	0.03329645

Joined with custom function

```
(tc/pivot->wider production ["product" "country"] "production" {:concat-columns-with vec})
```

data/production.csv [15 4]:

year	[“A” “AI”]	[“B” “AI”]	[“B” “EI”]
2000	1.63727158	-0.02617661	1.40470848
2001	0.15870784	-0.68863576	-0.59618369
2002	-1.56797745	0.06248741	-0.26568579
2003	-0.44455509	-0.72339686	0.65257808
2004	-0.07133701	0.47248952	0.62564999
2005	1.61183090	-0.94173861	-1.34530299
2006	-0.70434682	-0.34782108	-0.97184975
2007	-1.53550542	0.52425284	-1.69715821
2008	0.83907155	1.83230937	0.04556128
2009	-0.37424110	0.10706491	1.19315043
2010	-0.71158926	-0.32903664	-1.60557503
2011	1.12805634	-1.78319121	-0.77235497
2012	1.45718247	0.61125798	-2.50262738
2013	-1.55934101	-0.78526092	-1.62753769
2014	-0.11695838	0.97843635	0.03329645

Multiple value columns

```
(def income (tc/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

GEOID	NAME	variable	estimate	moe
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
...	...	...	...	...
51	Virginia	rent	1166	5
53	Washington	income	32318	113
53	Washington	rent	1120	4
54	West Virginia	income	23707	203
54	West Virginia	rent	681	6
55	Wisconsin	income	29868	135
55	Wisconsin	rent	813	3
56	Wyoming	income	30854	342
56	Wyoming	rent	828	11
72	Puerto Rico	income		
72	Puerto Rico	rent	464	6

```
(tc/pivot->wider income "variable" ["estimate" "moe"] {:drop-missing? false})
```

data/us_rent_income.csv [52 6]:

GEOID	NAME	income-estimate	income-moe	rent-estimate	rent-moe
1	Alabama	24476	136	747	3
2	Alaska	32940	508	1200	13
4	Arizona	27517	148	972	4
5	Arkansas	23789	165	709	5
6	California	29454	109	1358	3
8	Colorado	32401	109	1125	5
9	Connecticut	35326	195	1123	5
10	Delaware	31560	247	1076	10
11	District of Columbia	43198	681	1424	17
12	Florida	25952	70	1077	3
...	...	...	...	...	...
46	South Dakota	28821	276	696	7
47	Tennessee	25453	102	808	4
48	Texas	28063	110	952	2
49	Utah	27928	239	948	6
50	Vermont	29351	361	945	11
51	Virginia	32545	202	1166	5
53	Washington	32318	113	1120	4
54	West Virginia	23707	203	681	6
55	Wisconsin	29868	135	813	3
56	Wyoming	30854	342	828	11
72	Puerto Rico			464	6

Value concatenated by custom function

```
(tc/pivot->wider income "variable" ["estimate" "moe"] {:concat-columns-with vec
                                                       :concat-value-with vector
                                                       :drop-missing? false})
```

data/us_rent_income.csv [52 6]:

GEOID	NAME	[“income” “estimate”]	[“income” “moe”]	[“rent” “estimate”]	[“rent” “moe”]
1	Alabama	24476	136	747	3
2	Alaska	32940	508	1200	13
4	Arizona	27517	148	972	4
5	Arkansas	23789	165	709	5
6	California	29454	109	1358	3
8	Colorado	32401	109	1125	5
9	Connecticut	35326	195	1123	5
10	Delaware	31560	247	1076	10
11	District of Columbia	43198	681	1424	17
12	Florida	25952	70	1077	3
...	...	...	...	...	...
46	South Dakota	28821	276	696	7
47	Tennessee	25453	102	808	4
48	Texas	28063	110	952	2
49	Utah	27928	239	948	6
50	Vermont	29351	361	945	11
51	Virginia	32545	202	1166	5
53	Washington	32318	113	1120	4
54	West Virginia	23707	203	681	6
55	Wisconsin	29868	135	813	3
56	Wyoming	30854	342	828	11
72	Puerto Rico			464	6

Reshape contact data

```
(def contacts (tc/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

```
(tc/pivot->wider contacts "field" "value" {:drop-missing? false})
```

data/contacts.csv [3 4]:

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

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

World bank

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

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

data/world_bank_pop.csv.gz [1056 8]:

country	indicator	2000	2001	2002	2003	2004	2005
ABW	SP.URB.TOTL	4.24440000E+04	4.30480000E+04	4.36700000E+04	4.42460000E+04	4.46690000E+04	4.48890000E+04
ABW	SP.URB.GROW	1.78263237E+01	1.41302122E+01	1.43455953E+01	1.31036044E+01	9.51477684E-01	4.91302715E-01
ABW	SP.POP.TOTL	9.08530000E+04	9.28980000E+04	9.49920000E+04	9.70170000E+04	9.87370000E+04	1.00031000E+05
ABW	SP.POP.GROW	2.25502678E+00	2.22593013E+00	2.22905605E+00	2.10935434E+00	1.75735287E+00	1.30203884E+00
AFG	SP.URB.TOTL	4.43629900E+06	4.64805500E+06	4.89295100E+06	5.15568600E+06	5.42677000E+06	5.69182300E+06
AFG	SP.URB.GROW	3.91222846E+00	4.66283822E+00	5.13467454E+00	5.23045853E+00	5.12439302E+00	4.76864700E+00
AFG	SP.POP.TOTL	2.10937560E+07	2.09664630E+07	2.19799230E+07	2.30648510E+07	2.41189790E+07	2.50707980E+07
AFG	SP.POP.GROW	3.49465874E+00	4.25150411E+00	4.72052846E+00	4.81804112E+00	4.46891840E+00	3.87047016E+00
AGO	SP.URB.TOTL	8.23476600E+06	8.70800000E+06	9.21878700E+06	9.76519700E+06	1.03435060E+07	1.09494240E+07
AGO	SP.URB.GROW	5.43749411E+00	5.58771954E+00	5.70013237E+00	5.75812711E+00	5.75341450E+00	5.69279690E+00
...	...	...	...	...	...	...	...
ZAF	SP.URB.GROW	2.32229180E+00	2.26080492E+00	2.29242659E+00	2.25719919E+00	2.18014731E+00	2.09725981E+00
ZAF	SP.POP.TOTL	4.57283150E+07	4.63850060E+07	4.70261730E+07	4.76487270E+07	4.82473950E+07	4.88205860E+07
ZAF	SP.POP.GROW	1.47499416E+00	1.42585702E+00	1.37280586E+00	1.31515951E+00	1.24859226E+00	1.18102315E+00
ZMB	SP.URB.TOTL	3.66507600E+06	3.78866000E+06	3.94496500E+06	4.10631700E+06	4.27387500E+06	4.44857100E+06
ZMB	SP.URB.GROW	3.30532147E+00	3.31633227E+00	4.04276877E+00	4.00864374E+00	3.99943902E+00	4.00620111E+00
ZMB	SP.POP.TOTL	1.05312210E+07	1.08241250E+07	1.11204090E+07	1.14219840E+07	1.17317460E+07	1.20521560E+07
ZMB	SP.POP.GROW	2.80705843E+00	2.74331654E+00	2.70046295E+00	2.67578507E+00	2.67585813E+00	2.69450644E+00
ZWE	SP.URB.TOTL	4.12598700E+06	4.22551900E+06	4.32330700E+06	4.35604100E+06	4.38192000E+06	4.41384500E+06
ZWE	SP.URB.GROW	2.52373518E+00	2.38368296E+00	2.28785252E+00	7.54299867E-01	5.92336717E-01	7.25920717E-01
ZWE	SP.POP.TOTL	1.22222510E+07	1.23661650E+07	1.25005250E+07	1.26338970E+07	1.27775110E+07	1.29400320E+07
ZWE	SP.POP.GROW	1.29878201E+00	1.17059711E+00	1.08065293E+00	1.06127964E+00	1.13032327E+00	1.26390895E+00

Step 1 - convert years column into values

```
(def pop2 (tc/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	4.43600000E+04
ABW	SP.URB.GROW	2013	6.69503994E-01
ABW	SP.POP.TOTL	2013	1.03187000E+05
ABW	SP.POP.GROW	2013	5.92914005E-01
AFG	SP.URB.TOTL	2013	7.73396400E+06
AFG	SP.URB.GROW	2013	4.19297967E+00
AFG	SP.POP.TOTL	2013	3.17316880E+07
AFG	SP.POP.GROW	2013	3.31522413E+00
AGO	SP.URB.TOTL	2013	1.61194910E+07
AGO	SP.URB.GROW	2013	4.72272270E+00
...	...	...	...
ZAF	SP.URB.GROW	2012	2.23077040E+00
ZAF	SP.POP.TOTL	2012	5.29982130E+07
ZAF	SP.POP.GROW	2012	1.39596592E+00
ZMB	SP.URB.TOTL	2012	5.93201300E+06
ZMB	SP.URB.GROW	2012	4.25944078E+00
ZMB	SP.POP.TOTL	2012	1.46999370E+07
ZMB	SP.POP.GROW	2012	3.00513283E+00
ZWE	SP.URB.TOTL	2012	4.83015300E+06
ZWE	SP.URB.GROW	2012	1.67857380E+00
ZWE	SP.POP.TOTL	2012	1.47108260E+07
ZWE	SP.POP.GROW	2012	2.22830616E+00

Step 2 - separate `"indicate"` column

```
(def pop3 (tc/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	4.43600000E+04
ABW	URB	GROW	2013	6.69503994E-01
ABW	POP	TOTL	2013	1.03187000E+05
ABW	POP	GROW	2013	5.92914005E-01
AFG	URB	TOTL	2013	7.73396400E+06
AFG	URB	GROW	2013	4.19297967E+00
AFG	POP	TOTL	2013	3.17316880E+07
AFG	POP	GROW	2013	3.31522413E+00
AGO	URB	TOTL	2013	1.61194910E+07
AGO	URB	GROW	2013	4.72272270E+00
...	...	...	...	...
ZAF	URB	GROW	2012	2.23077040E+00
ZAF	POP	TOTL	2012	5.29982130E+07

country	area	variable	year	value
ZAF	POP	GROW	2012	1.39596592E+00
ZMB	URB	TOTL	2012	5.93201300E+06
ZMB	URB	GROW	2012	4.25944078E+00
ZMB	POP	TOTL	2012	1.46999370E+07
ZMB	POP	GROW	2012	3.00513283E+00
ZWE	URB	TOTL	2012	4.83015300E+06
ZWE	URB	GROW	2012	1.67857380E+00
ZWE	POP	TOTL	2012	1.47108260E+07
ZWE	POP	GROW	2012	2.22830616E+00

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

```
(tc/pivot->wider pop3 "variable" "value" {:drop-missing? false})
```

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

country	area	year	TOTL	GROW
ABW	URB	2013	4.43600000E+04	0.66950399
ABW	POP	2013	1.03187000E+05	0.59291401
AFG	URB	2013	7.73396400E+06	4.19297967
AFG	POP	2013	3.17316880E+07	3.31522413
AGO	URB	2013	1.61194910E+07	4.72272270
AGO	POP	2013	2.59983400E+07	3.53182419
ALB	URB	2013	1.60350500E+06	1.74363937
ALB	POP	2013	2.89509200E+06	-0.18321138
AND	URB	2013	7.15270000E+04	-2.11923331
AND	POP	2013	8.07880000E+04	-2.01331401
...	...	...	...	...
WSM	POP	2012	1.89194000E+05	0.81144852
XKX	URB	2012		
XKX	POP	2012	1.80520000E+06	0.78972659
YEM	URB	2012	8.20982800E+06	4.49478765
YEM	POP	2012	2.49099690E+07	2.67605025
ZAF	URB	2012	3.35330290E+07	2.23077040
ZAF	POP	2012	5.29982130E+07	1.39596592
ZMB	URB	2012	5.93201300E+06	4.25944078
ZMB	POP	2012	1.46999370E+07	3.00513283
ZWE	URB	2012	4.83015300E+06	1.67857380
ZWE	POP	2012	1.47108260E+07	2.22830616

Multi-choice

```
(def multi (tc/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
                (tc/pivot->longer (complement #{:id}))
                (tc/add-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
    (tc/drop-columns :$column)
    (tc/pivot->wider :$value :checked {:drop-missing? false})
    (tc/order-by :id))
```

_unnamed [4 5]:

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

Construction

```
(def construction (tc/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
    (tc/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	August		South	649
2018	September		South	560
2018	January		West	339
2018	February		West	336
2018	March		West	330
2018	April		West	304
2018	May		West	319
2018	June		West	360
2018	July		West	310
2018	August		West	286
2018	September		West	296

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

```
(-> construction
    (tc/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})
    (tc/pivot->wider [:units :region] :n {:drop-missing? false})
    (tc/rename-columns (zipmap (vals construction-unit-map)
                               (keys construction-unit-map))))
```

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

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

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

```
stocks-tidyr
```

data/stockstidyr.csv [10 4]:

time	X	Y	Z
2009-01-01	1.30989806	-1.89040193	-1.77946880
2009-01-02	-0.29993804	-1.82473090	2.39892513
2009-01-03	0.53647501	-1.03606860	-3.98697977
2009-01-04	-1.88390802	-0.52178390	-2.83065490
2009-01-05	-0.96052361	-2.21683349	1.43715171
2009-01-06	-1.18528966	-2.89350924	3.39784140
2009-01-07	-0.85207056	-2.16794818	-1.20108258
2009-01-08	0.25234172	-0.32854117	-1.53160473
2009-01-09	0.40257136	1.96407898	-6.80878830
2009-01-10	-0.64383500	2.68618382	-2.55909321

Convert to longer form

```
(def stocks-long (tc/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	1.30989806
2009-01-02	X	-0.29993804
2009-01-03	X	0.53647501
2009-01-04	X	-1.88390802
2009-01-05	X	-0.96052361
2009-01-06	X	-1.18528966
2009-01-07	X	-0.85207056
2009-01-08	X	0.25234172
2009-01-09	X	0.40257136
2009-01-10	X	-0.64383500
...	...	...
2009-01-10	Y	2.68618382
2009-01-01	Z	-1.77946880
2009-01-02	Z	2.39892513
2009-01-03	Z	-3.98697977
2009-01-04	Z	-2.83065490
2009-01-05	Z	1.43715171
2009-01-06	Z	3.39784140
2009-01-07	Z	-1.20108258
2009-01-08	Z	-1.53160473
2009-01-09	Z	-6.80878830
2009-01-10	Z	-2.55909321

Convert back to wide form

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

data/stockstidyr.csv [10 4]:

time	X	Y	Z
2009-01-01	1.30989806	-1.89040193	-1.77946880
2009-01-02	-0.29993804	-1.82473090	2.39892513
2009-01-03	0.53647501	-1.03606860	-3.98697977
2009-01-04	-1.88390802	-0.52178390	-2.83065490
2009-01-05	-0.96052361	-2.21683349	1.43715171
2009-01-06	-1.18528966	-2.89350924	3.39784140
2009-01-07	-0.85207056	-2.16794818	-1.20108258
2009-01-08	0.25234172	-0.32854117	-1.53160473
2009-01-09	0.40257136	1.96407898	-6.80878830
2009-01-10	-0.64383500	2.68618382	-2.55909321

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

```
(-> stocks-long
    (tc/select-rows (range 0 30 4))
    (tc/pivot->wider "time" :price {:drop-missing? false}))
```

data/stockstidyr.csv [3 6]:

:stocks	2009-01-01	2009-01-05	2009-01-09	2009-01-03	2009-01-07
Y				-1.0360686	-2.16794818
X	1.30989806	-0.96052361	0.40257136		
Z	-1.77946880	1.43715171	-6.80878830		

Join/Concat Datasets

Dataset join and concatenation functions.

Joins accept left-side and right-side datasets and columns selector. Options are the same as in `tech.ml.dataset` functions.

A column selector can be a map with `:left` and `:right` keys to specify column names separate for left and right dataset.

The difference between `tech.ml.dataset` join functions are: arguments order (first datasets) and possibility to join on multiple columns.

Additionally set operations are defined: `intersect` and `difference`.

To concat two datasets rowwise you can choose:

- `concat` - concats rows for matching columns, the number of columns should be equal.
- `union` - like concat but returns unique values
- `bind` - concats rows add missing, empty columns

To add two datasets columnwise use `bind`. The number of rows should be equal.

Datasets used in examples:

```
(def ds1 (tc/dataset {:a [1 2 1 2 3 4 nil nil 4]
                      :b (range 101 110)
                      :c (map str "abs tract")}))
(def ds2 (tc/dataset {:a [nil 1 2 5 4 3 2 1 nil]
                      :b (range 110 101 -1)
                      :c (map str "datatable")
                      :d (symbol "X")
                      :e [3 4 5 6 7 nil 8 1 1]}))
```

```
ds1
ds2
```

_unnamed [9 3]:

:a	:b	:c
1	101	a
2	102	b
1	103	s
2	104	
3	105	t
4	106	r
	107	a
	108	c
4	109	t

_unnamed [9 5]:

:a	:b	:c	:d	:e
	110	d	X	3
1	109	a	X	4
2	108	t	X	5
5	107	a	X	6
4	106	t	X	7
3	105	a	X	
2	104	b	X	8
1	103	l	X	1
	102	e	X	1

```
(tc/left-join ds1 ds2 :b)
```

Left left-outer-join [9 8]:

:b	:a	:c	:right.b	:right.a	:right.c	:d	:e
109	4	t	109	1	a	X	4
108		c	108	2	t	X	5
107		a	107	5	a	X	6
106	4	r	106	4	t	X	7
105	3	t	105	3	a	X	
104	2		104	2	b	X	8
103	1	s	103	1	l	X	1
102	2	b	102		e	X	1
101	1	a					

```
(tc/left-join ds2 ds1 :b)
```

left-outer-join [9 8]:

:b	:a	:c	:d	:e	:right.b	:right.a	:right.c
102		e	X	1	102	2	b
103	1	l	X	1	103	1	s
104	2	b	X	8	104	2	
105	3	a	X		105	3	t
106	4	t	X	7	106	4	r
107	5	a	X	6	107		a
108	2	t	X	5	108		c
109	1	a	X	4	109	4	t
110		d	X	3			

```
(tc/left-join ds1 ds2 [:a :b])
```

left-outer-join [9 8]:

:a	:b	:c	:right.a	:right.b	:right.c	:d	:e
4	106	r	4	106	t	X	7
3	105	t	3	105	a	X	
2	104		2	104	b	X	8
1	103	s	1	103	l	X	1
1	101	a					
2	102	b					
	107	a					
	108	c					
4	109	t					

```
(tc/left-join ds2 ds1 [:a :b])
```

left-outer-join [9 8]:

:a	:b	:c	:d	:e	:right.a	:right.b	:right.c
1	103	l	X	1	1	103	s
2	104	b	X	8	2	104	
3	105	a	X		3	105	t
4	106	t	X	7	4	106	r
	110	d	X	3			
1	109	a	X	4			
2	108	t	X	5			
5	107	a	X	6			
	102	e	X	1			

```
(tc/left-join ds1 ds2 {:left :a :right :e})
```

left-outer-join [11 8]:

:a	:b	:c	:e	:right.a	:right.b	:right.c	:d
3	105	t	3		110	d	X
4	106	r	4	1	109	a	X
4	109	t	4	1	109	a	X
	107	a		3	105	a	X
	108	c		3	105	a	X
1	101	a	1	1	103	l	X
1	103	s	1	1	103	l	X
1	101	a	1		102	e	X
1	103	s	1		102	e	X
2	102	b					
2	104						

```
(tc/left-join ds2 ds1 {:left :e :right :a})
```

left-outer-join [13 8]:

:e	:a	:b	:c	:d	:right.a	:right.b	:right.c
1	1	103	l	X	1	101	a
1		102	e	X	1	101	a
1	1	103	l	X	1	103	s
1		102	e	X	1	103	s
3		110	d	X	3	105	t
4	1	109	a	X	4	106	r
	3	105	a	X		107	a
	3	105	a	X		108	c
4	1	109	a	X	4	109	t
5	2	108	t	X			
6	5	107	a	X			
7	4	106	t	X			
8	2	104	b	X			

```
(tc/right-join ds1 ds2 :b)
```

Right right-outer-join [9 8]:

:b	:a	:c	:right.b	:right.a	:right.c	:d	:e
109	4	t	109	1	a	X	4
108		c	108	2	t	X	5
107		a	107	5	a	X	6
106	4	r	106	4	t	X	7
105	3	t	105	3	a	X	
104	2		104	2	b	X	8
103	1	s	103	1	l	X	1
102	2	b	102		e	X	1
			110		d	X	3

```
(tc/right-join ds2 ds1 :b)
```

right-outer-join [9 8]:

:b	:a	:c	:d	:e	:right.b	:right.a	:right.c
102		e	X	1	102	2	b
103	1	l	X	1	103	1	s
104	2	b	X	8	104	2	
105	3	a	X		105	3	t
106	4	t	X	7	106	4	r
107	5	a	X	6	107		a
108	2	t	X	5	108		c
109	1	a	X	4	109	4	t
					101	1	a

```
(tc/right-join ds1 ds2 [:a :b])
```

right-outer-join [9 8]:

:a	:b	:c	:right.a	:right.b	:right.c	:d	:e
4	106	r	4	106	t	X	7
3	105	t	3	105	a	X	
2	104		2	104	b	X	8
1	103	s	1	103	l	X	1
				110	d	X	3
			1	109	a	X	4
			2	108	t	X	5
			5	107	a	X	6
				102	e	X	1

```
(tc/right-join ds2 ds1 [:a :b])
```

right-outer-join [9 8]:

:a	:b	:c	:d	:e	:right.a	:right.b	:right.c
1	103	l	X	1	1	103	s
2	104	b	X	8	2	104	
3	105	a	X		3	105	t
4	106	t	X	7	4	106	r
					1	101	a
					2	102	b
						107	a
						108	c
					4	109	t

```
(tc/right-join ds1 ds2 {:left :a :right :e})
```

right-outer-join [13 8]:

:a	:b	:c	:e	:right.a	:right.b	:right.c	:d
3	105	t	3		110	d	X
4	106	r	4	1	109	a	X
4	109	t	4	1	109	a	X
	107	a		3	105	a	X
	108	c		3	105	a	X
1	101	a	1	1	103	l	X
1	103	s	1	1	103	l	X
1	101	a	1		102	e	X
1	103	s	1		102	e	X
			5	2	108	t	X
			6	5	107	a	X
			7	4	106	t	X
			8	2	104	b	X

```
(tc/right-join ds2 ds1 {:left :e :right :a})
```

right-outer-join [11 8]:

:e	:a	:b	:c	:d	:right.a	:right.b	:right.c
1	1	103	l	X	1	101	a
1		102	e	X	1	101	a
1	1	103	l	X	1	103	s
1		102	e	X	1	103	s
3		110	d	X	3	105	t
4	1	109	a	X	4	106	r
	3	105	a	X		107	a
	3	105	a	X		108	c
4	1	109	a	X	4	109	t
					2	102	b
					2	104	

```
(tc/inner-join ds1 ds2 :b)
```

Inner inner-join [8 7]:

:b	:a	:c	:right.a	:right.c	:d	:e
109	4	t	1	a	X	4
108		c	2	t	X	5
107		a	5	a	X	6
106	4	r	4	t	X	7
105	3	t	3	a	X	
104	2		2	b	X	8
103	1	s	1	l	X	1
102	2	b		e	X	1

```
(tc/inner-join ds2 ds1 :b)
```

inner-join [8 7]:

:b	:a	:c	:d	:e	:right.a	:right.c
102		e	X	1	2	b
103	1	l	X	1	1	s
104	2	b	X	8	2	
105	3	a	X		3	t
106	4	t	X	7	4	r
107	5	a	X	6		a
108	2	t	X	5		c
109	1	a	X	4	4	t

```
(tc/inner-join ds1 ds2 [:a :b])
```

inner-join [4 8]:

:a	:b	:c	:right.a	:right.b	:right.c	:d	:e
4	106	r	4	106	t	X	7
3	105	t	3	105	a	X	
2	104		2	104	b	X	8
1	103	s	1	103	l	X	1

```
(tc/inner-join ds2 ds1 [:a :b])
```

inner-join [4 8]:

:a	:b	:c	:d	:e	:right.a	:right.b	:right.c
1	103	l	X	1	1	103	s
2	104	b	X	8	2	104	
3	105	a	X		3	105	t
4	106	t	X	7	4	106	r

```
(tc/inner-join ds1 ds2 {:left :a :right :e})
```

inner-join [9 7]:

:a	:b	:c	:right.a	:right.b	:right.c	:d
3	105	t		110	d	X
4	106	r	1	109	a	X
4	109	t	1	109	a	X
	107	a	3	105	a	X
	108	c	3	105	a	X
1	101	a	1	103	l	X
1	103	s	1	103	l	X
1	101	a		102	e	X
1	103	s		102	e	X

```
(tc/inner-join ds2 ds1 {:left :e :right :a})
```

inner-join [9 7]:

:e	:a	:b	:c	:d	:right.b	:right.c
1	1	103	l	X	101	a
1		102	e	X	101	a
1	1	103	l	X	103	s
1		102	e	X	103	s
3		110	d	X	105	t
4	1	109	a	X	106	r

:e	:a	:b	:c	:d	:right.b	:right.c
	3	105	a	X	107	a
	3	105	a	X	108	c
4	1	109	a	X	109	t

Full Join keeping all rows

```
(tc/full-join ds1 ds2 :b)
```

full-join [10 8]:

:b	:a	:c	:right.b	:right.a	:right.c	:d	:e
109	4	t	109	1	a	X	4
108		c	108	2	t	X	5
107		a	107	5	a	X	6
106	4	r	106	4	t	X	7
105	3	t	105	3	a	X	
104	2		104	2	b	X	8
103	1	s	103	1	l	X	1
102	2	b	102		e	X	1
101	1	a					
			110		d	X	3

```
(tc/full-join ds2 ds1 :b)
```

full-join [10 8]:

:b	:a	:c	:d	:e	:right.b	:right.a	:right.c
102		e	X	1	102	2	b
103	1	l	X	1	103	1	s
104	2	b	X	8	104	2	
105	3	a	X		105	3	t
106	4	t	X	7	106	4	r
107	5	a	X	6	107		a
108	2	t	X	5	108		c
109	1	a	X	4	109	4	t
110		d	X	3			
					101	1	a

```
(tc/full-join ds1 ds2 [:a :b])
```

full-join [14 8]:

:a	:b	:c	:right.a	:right.b	:right.c	:d	:e
4	106	r	4	106	t	X	7
3	105	t	3	105	a	X	
2	104		2	104	b	X	8
1	103	s	1	103	l	X	1

:a	:b	:c	:right.a	:right.b	:right.c	:d	:e
1	101	a					
2	102	b					
	107	a					
	108	c					
4	109	t					
				110	d	X	3
			1	109	a	X	4
			2	108	t	X	5
			5	107	a	X	6
				102	e	X	1

```
(tc/full-join ds2 ds1 [:a :b])
```

full-join [14 8]:

:a	:b	:c	:d	:e	:right.a	:right.b	:right.c
1	103	l	X	1	1	103	s
2	104	b	X	8	2	104	
3	105	a	X		3	105	t
4	106	t	X	7	4	106	r
	110	d	X	3			
1	109	a	X	4			
2	108	t	X	5			
5	107	a	X	6			
	102	e	X	1			
					1	101	a
					2	102	b
						107	a
						108	c
					4	109	t

```
(tc/full-join ds1 ds2 {:left :a :right :e})
```

full-join [15 8]:

:a	:b	:c	:e	:right.a	:right.b	:right.c	:d
3	105	t	3		110	d	X
4	106	r	4	1	109	a	X
4	109	t	4	1	109	a	X
	107	a		3	105	a	X
	108	c		3	105	a	X
1	101	a	1	1	103	l	X
1	103	s	1	1	103	l	X
1	101	a	1		102	e	X
1	103	s	1		102	e	X
2	102	b					
2	104						

:a	:b	:c	:e	:right.a	:right.b	:right.c	:d
			5	2	108	t	X
			6	5	107	a	X
			7	4	106	t	X
			8	2	104	b	X

```
(tc/full-join ds2 ds1 {:left :e :right :a})
```

full-join [15 8]:

:e	:a	:b	:c	:d	:right.a	:right.b	:right.c
1	1	103	l	X	1	101	a
1		102	e	X	1	101	a
1	1	103	l	X	1	103	s
1		102	e	X	1	103	s
3		110	d	X	3	105	t
4	1	109	a	X	4	106	r
	3	105	a	X		107	a
	3	105	a	X		108	c
4	1	109	a	X	4	109	t
5	2	108	t	X			
6	5	107	a	X			
7	4	106	t	X			
8	2	104	b	X			
					2	102	b
					2	104	

Semi Return rows from ds1 matching ds2

```
(tc/semi-join ds1 ds2 :b)
```

semi-join [4 3]:

:b	:a	:c
109	4	t
106	4	r
104	2	
103	1	s

```
(tc/semi-join ds2 ds1 :b)
```

semi-join [4 5]:

:b	:a	:c	:d	:e
103	1	l	X	1
104	2	b	X	8
106	4	t	X	7

:b	:a	:c	:d	:e
109	1	a	X	4

```
(tc/semi-join ds1 ds2 [:a :b])
```

semi-join [3 3]:

:a	:b	:c
4	106	r
2	104	
1	103	s

```
(tc/semi-join ds2 ds1 [:a :b])
```

semi-join [3 5]:

:a	:b	:c	:d	:e
1	103	l	X	1
2	104	b	X	8
4	106	t	X	7

```
(tc/semi-join ds1 ds2 {:left :a :right :e})
```

semi-join [4 3]:

:a	:b	:c
4	106	r
4	109	t
1	101	a
1	103	s

```
(tc/semi-join ds2 ds1 {:left :e :right :a})
```

semi-join [2 5]:

:e	:a	:b	:c	:d
1	1	103	l	X
4	1	109	a	X

Anti Return rows from ds1 not matching ds2

```
(tc/anti-join ds1 ds2 :b)
```

anti-join [5 3]:

:b	:a	:c
108		c
107		a
105	3	t
102	2	b
101	1	a

```
(tc/anti-join ds2 ds1 :b)
```

anti-join [5 5]:

:b	:a	:c	:d	:e
102		e	X	1
105	3	a	X	
107	5	a	X	6
108	2	t	X	5
110		d	X	3

```
(tc/anti-join ds1 ds2 [:a :b])
```

anti-join [6 3]:

:a	:b	:c
3	105	t
1	101	a
2	102	b
	107	a
	108	c
4	109	t

```
(tc/anti-join ds1 ds2 {:left :a :right :e})
```

anti-join [7 3]:

:a	:b	:c
3	105	t
	107	a
	108	c
1	101	a
1	103	s
2	102	b
2	104	

```
(tc/anti-join ds2 ds1 {:left :e :right :a})
```

anti-join [7 5]:

:e	:a	:b	:c	:d
1		102	e	X
3		110	d	X
	3	105	a	X
5	2	108	t	X
6	5	107	a	X
7	4	106	t	X
8	2	104	b	X

Cross Cross product from selected columns

```
(tc/cross-join ds1 ds2 [:a :b])
```

cross-join [81 4]:

:a	:b	:right.a	:right.b
1	101		110
1	101	1	109
1	101	2	108
1	101	5	107
1	101	4	106
1	101	3	105
1	101	2	104
1	101	1	103
1	101		102
2	102		110
...	...	...	...
	108	1	103
	108		102
4	109		110
4	109	1	109
4	109	2	108
4	109	5	107
4	109	4	106
4	109	3	105
4	109	2	104
4	109	1	103
4	109		102

```
(tc/cross-join ds1 ds2 {:left [:a :b] :right :e})
```

cross-join [81 3]:

:a	:b	:e
1	101	3
1	101	4

:a	:b	:e
1	101	5
1	101	6
1	101	7
1	101	
1	101	8
1	101	1
1	101	1
2	102	3
...	...	...
	108	1
	108	1
4	109	3
4	109	4
4	109	5
4	109	6
4	109	7
4	109	
4	109	8
4	109	1
4	109	1

Expand Similar to cross product but works on a single dataset.

```
(tc/expand ds2 :a :c :d)
```

cross-join [36 3]:

:a	:c	:d
	d	X
	a	X
	t	X
	b	X
	l	X
	e	X
1	d	X
1	a	X
1	t	X
1	b	X
...	...	...
4	a	X
4	t	X
4	b	X
4	l	X
4	e	X
3	d	X
3	a	X
3	t	X
3	b	X
3	l	X
3	e	X

Columns can be also bundled (nested) in tuples which are treated as a single entity during cross product.

```
(tc/expand ds2 [:a :c] [:e :b])
```

cross-join [81 4]:

:a	:c	:e	:b
	d	3	110
	d	4	109
	d	5	108
	d	6	107
	d	7	106
	d		105
	d	8	104
	d	1	103
	d	1	102
1	a	3	110
...	...	...	...
1	l	1	103
1	l	1	102
	e	3	110
	e	4	109
	e	5	108
	e	6	107
	e	7	106
	e		105
	e	8	104
	e	1	103
	e	1	102

Complete Same as expand with all other columns preserved (filled with missing values if necessary).

```
(tc/complete ds2 :a :c :d)
```

left-outer-join [36 5]:

:a	:c	:d	:b	:e
	d	X	110	3
1	a	X	109	4
2	t	X	108	5
5	a	X	107	6
4	t	X	106	7
3	a	X	105	
2	b	X	104	8
1	l	X	103	1
	e	X	102	1
	a	X		
...	...	...	...	...
5	e	X		
4	d	X		
4	a	X		
4	b	X		
4	l	X		

:a	:c	:d	:b	:e
4	e	X		
3	d	X		
3	t	X		
3	b	X		
3	l	X		
3	e	X		

```
(tc/complete ds2 [:a :c] [:e :b])
```

left-outer-join [81 5]:

:a	:c	:e	:b	:d
	d	3	110	X
1	a	4	109	X
2	t	5	108	X
5	a	6	107	X
4	t	7	106	X
3	a		105	X
2	b	8	104	X
1	l	1	103	X
	e	1	102	X
	d	4	109	
...	...	...	...	...
1	l		105	
1	l	8	104	
1	l	1	102	
	e	3	110	
	e	4	109	
	e	5	108	
	e	6	107	
	e	7	106	
	e		105	
	e	8	104	
	e	1	103	

```
(def left-ds (tc/dataset {:a [1 5 10]
                          :left-val ["a" "b" "c"]}))
(def right-ds (tc/dataset {:a [1 2 3 6 7]
                           :right-val [:a :b :c :d :e]}))
```

```
left-ds
right-ds
```

asof _unnamed [3 2]:

:a	:left-val
1	a

:a	:left-val
5	b
10	c

_unnamed [5 2]:

:a	:right-val
1	:a
2	:b
3	:c
6	:d
7	:e

```
(tc/asof-join left-ds right-ds :a)
```

asof-<= [3 4]:

:a	:left-val	:right.a	:right-val
1	a	1	:a
5	b	6	:d
10	c		

```
(tc/asof-join left-ds right-ds :a {:asof-op :nearest})
```

asof-nearest [3 4]:

:a	:left-val	:right.a	:right-val
1	a	1	:a
5	b	6	:d
10	c	7	:e

```
(tc/asof-join left-ds right-ds :a {:asof-op :>=})
```

asof->= [3 4]:

:a	:left-val	:right.a	:right-val
1	a	1	:a
5	b	3	:c
10	c	7	:e

Concat `contact` joins rows from other datasets

```
(tc/concat ds1)
```

_unnamed [9 3]:

:a	:b	:c
1	101	a
2	102	b
1	103	s
2	104	
3	105	t
4	106	r
	107	a
	108	c
4	109	t

`concat-copying` ensures all readers are evaluated.

```
(tc/concat-copying ds1)
```

_unnamed [9 3]:

:a	:b	:c
1	101	a
2	102	b
1	103	s
2	104	
3	105	t
4	106	r
	107	a
	108	c
4	109	t

```
(tc/concat ds1 (tc/drop-columns ds2 :d))
```

_unnamed [18 4]:

:a	:b	:c	:e
1	101	a	
2	102	b	
1	103	s	
2	104		
3	105	t	
4	106	r	
	107	a	
	108	c	
4	109	t	
	110	d	3
1	109	a	4
2	108	t	5
5	107	a	6
4	106	t	7
3	105	a	
2	104	b	8

:a	:b	:c	:e
1	103	l	1
	102	e	1

```
(apply tc/concat (repeatedly 3 #(tc/random DS)))
```

_unnamed [27 4]:

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

Concat grouped dataset Concatenation of grouped datasets results also in grouped dataset.

```
(tc/concat (tc/group-by DS [:V3])
           (tc/group-by DS [:V4]))
```

_unnamed [6 3]:

:group-id	:name	:data
0	{:V3 0.5}	Group: {:V3 0.5} [3 4]:
1	{:V3 1.0}	Group: {:V3 1.0} [3 4]:
2	{:V3 1.5}	Group: {:V3 1.5} [3 4]:
3	{:V4 “A”}	Group: {:V4 “A”} [3 4]:
4	{:V4 “B”}	Group: {:V4 “B”} [3 4]:
5	{:V4 “C”}	Group: {:V4 “C”} [3 4]:

Union The same as `concat` but returns unique rows

```
(apply tc/union (tc/drop-columns ds2 :d) (repeat 10 ds1))
```

union [18 4]:

:a	:b	:c	:e
	110	d	3
1	109	a	4
2	108	t	5
5	107	a	6
4	106	t	7
3	105	a	
2	104	b	8
1	103	l	1
	102	e	1
1	101	a	
2	102	b	
1	103	s	
2	104		
3	105	t	
4	106	r	
	107	a	
	108	c	
4	109	t	

```
(apply tc/union (repeatedly 10 #(tc/random DS)))
```

union [9 4]:

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

Bind `bind` adds empty columns during concat

```
(tc/bind ds1 ds2)
```

_unnamed [18 5]:

:a	:b	:c	:e	:d
1	101	a		
2	102	b		
1	103	s		
2	104			

:a	:b	:c	:e	:d
3	105	t		
4	106	r		
	107	a		
	108	c		
4	109	t		
	110	d	3	X
1	109	a	4	X
2	108	t	5	X
5	107	a	6	X
4	106	t	7	X
3	105	a		X
2	104	b	8	X
1	103	l	1	X
	102	e	1	X

```
(tc/bind ds2 ds1)
```

_unnamed [18 5]:

:a	:b	:c	:d	:e
	110	d	X	3
1	109	a	X	4
2	108	t	X	5
5	107	a	X	6
4	106	t	X	7
3	105	a	X	
2	104	b	X	8
1	103	l	X	1
	102	e	X	1
1	101	a		
2	102	b		
1	103	s		
2	104			
3	105	t		
4	106	r		
	107	a		
	108	c		
4	109	t		

Append `append` concats columns

```
(tc/append ds1 ds2)
```

_unnamed [9 8]:

:a	:b	:c	:a	:b	:c	:d	:e
1	101	a		110	d	X	3
2	102	b	1	109	a	X	4
1	103	s	2	108	t	X	5

:a	:b	:c	:a	:b	:c	:d	:e
2	104		5	107	a	X	6
3	105	t	4	106	t	X	7
4	106	r	3	105	a	X	
	107	a	2	104	b	X	8
	108	c	1	103	l	X	1
4	109	t		102	e	X	1

```
(tc/intersect (tc/select-columns ds1 :b)
              (tc/select-columns ds2 :b))
```

Intersection intersection [8 1]:

:b
109
108
107
106
105
104
103
102

```
(tc/difference (tc/select-columns ds1 :b)
               (tc/select-columns ds2 :b))
```

Difference difference [1 1]:

:b
101

```
(tc/difference (tc/select-columns ds2 :b)
               (tc/select-columns ds1 :b))
```

difference [1 1]:

:b
110

Split into train/test

In ML world very often you need to test given model and prepare collection of train and test datasets. `split` creates new dataset with two additional columns:

- `:$split-name` - with `:train`, `:test`, `:split-2`, ... values
- `:$split-id` - id of splitted group (for k-fold and repeating)

`split-type` can be one of the following:

- `:kfold` (default) - k-fold strategy, `:k` defines number of folds (defaults to `5`), produces `k` splits
- `:bootstrap` - `:ratio` defines ratio of observations put into result (defaults to `1.0`), produces `1` split
- `:holdout` - split into two or more parts with given ratio(s) (defaults to `2/3`), produces `1` split
- `:holdouts` - splits into two parts for ascending ratio. Range of rations is given by `steps` option
- `:loo` - leave one out, produces the same number of splits as number of observations

`:holdout` can accept also probabilites or ratios and can split to more than 2 subdatasets

Additionally you can provide:

- `:seed` - for random number generator
- `:shuffle?` - turn on/off shuffle of the rows (default: `true`)
- `:repeats` - repeat procedure `:repeats` times
- `:partition-selector` - same as in `group-by` for stratified splitting to reflect dataset structure in splits.
- `:split-names` names of subdatasets different than default, ie. `[:train :test :split-2 ...]`
- `:split-col-name` - a column where name of split is stored, either `:train` or `:test` values (default: `:$split-name`)
- `:split-id-col-name` - a column where id of the train/test pair is stored (default: `:$split-id`)

In case of grouped dataset each group is processed separately.

See more

```
(def for-splitting (tc/dataset (map-indexed (fn [id v] {:id id
                                                        :partition v
                                                        :group (rand-nth [:g1 :g2 :g3])})
                                            (concat (repeat 20 :a) (repeat 5 :b)))))
```

```
for-splitting
```

_unnamed [25 3]:

:group	:partition	:id
:g1	:a	0
:g1	:a	1
:g1	:a	2
:g1	:a	3
:g2	:a	4
:g1	:a	5
:g1	:a	6
:g3	:a	7
:g2	:a	8
:g1	:a	9
...	...	...
:g2	:a	14
:g1	:a	15
:g3	:a	16
:g3	:a	17
:g1	:a	18
:g3	:a	19
:g2	:b	20
:g1	:b	21
:g2	:b	22
:g1	:b	23
:g3	:b	24

k-Fold Returns `k=5` maps

```
(-> for-splitting
    (tc/split)
    (tc/head 30))
```

_unnamed, (splitted) [30 5]:

:group	:partition	:id	:*split* − *name*| :split-id
:g3	:b	24	:train 0
:g2	:b	20	:train 0
:g1	:a	0	:train 0
:g3	:a	16	:train 0
:g1	:a	15	:train 0
:g1	:a	1	:train 0
:g3	:a	19	:train 0
:g1	:a	5	:train 0
:g1	:a	18	:train 0
:g2	:a	14	:train 0
:g1	:a	3	:train 0
:g1	:b	21	:train 0
:g3	:a	7	:train 0
:g1	:a	2	:train 0
:g2	:a	8	:train 0
:g1	:a	6	:train 0
:g1	:a	9	:train 0
:g3	:a	10	:train 0
:g1	:b	23	:train 0
:g1	:a	12	:train 0
:g3	:a	13	:test 0
:g2	:b	22	:test 0
:g2	:a	11	:test 0
:g2	:a	4	:test 0
:g3	:a	17	:test 0
:g3	:a	13	:train 1
:g2	:b	22	:train 1
:g2	:a	11	:train 1
:g2	:a	4	:train 1
:g3	:a	17	:train 1

Partition according to `:k` column to reflect it's distribution

```
(-> for-splitting
    (tc/split :kfold {:partition-selector :partition})
    (tc/head 30))
```

_unnamed, (splitted) [30 5]:

:group	:partition	:id	:*split* − *name*| :split-id
:g3	:a	13	:train 0
:g1	:a	2	:train 0
:g3	:a	7	:train 0
:g1	:a	3	:train 0

:group	:partition	:id	:*split − name*| :split-id	
:g2	:a	11	:train	0
:g1	:a	15	:train	0
:g1	:a	5	:train	0
:g2	:a	4	:train	0
:g1	:a	1	:train	0
:g1	:a	18	:train	0
:g1	:a	6	:train	0
:g3	:a	17	:train	0
:g3	:a	19	:train	0
:g1	:a	9	:train	0
:g1	:a	12	:train	0
:g3	:a	10	:train	0
:g2	:a	8	:test	0
:g1	:a	0	:test	0
:g2	:a	14	:test	0
:g3	:a	16	:test	0
:g2	:a	8	:train	1
:g1	:a	0	:train	1
:g2	:a	14	:train	1
:g3	:a	16	:train	1
:g2	:a	11	:train	1
:g1	:a	15	:train	1
:g1	:a	5	:train	1
:g2	:a	4	:train	1
:g1	:a	1	:train	1
:g1	:a	18	:train	1

```
(tc/split for-splitting :bootstrap)
```

Bootstrap _unnamed, (splitted) [34 5]:

:group	:partition	:id	:*split − name*| :split-id	
:g3	:a	17	:train	0
:g3	:a	7	:train	0
:g2	:a	8	:train	0
:g1	:a	2	:train	0
:g3	:a	7	:train	0
:g1	:a	5	:train	0
:g2	:a	8	:train	0
:g3	:a	19	:train	0
:g1	:a	6	:train	0
:g1	:a	12	:train	0
...	...	...	...	...
:g3	:a	16	:train	0
:g2	:b	20	:train	0
:g1	:a	1	:test	0
:g3	:b	24	:test	0
:g1	:a	15	:test	0
:g3	:a	13	:test	0
:g1	:a	3	:test	0

:group	:partition	:id	:*split* − *name*|	:split-id
:g2	:a	11	:test	0
:g1	:a	9	:test	0
:g3	:a	10	:test	0
:g1	:a	18	:test	0

with repeats, to get 100 splits

```
(-> for-splitting
    (tc/split :bootstrap {:repeats 100})
    (:$split-id)
    (distinct)
    (count))
```

100

Holdout with small ratio

```
(tc/split for-splitting :holdout {:ratio 0.2})
```

_unnamed, (splitted) [25 5]:

:group	:partition	:id	:*split* − *name*|	:split-id
:g3	:a	19	:train	0
:g2	:a	4	:train	0
:g1	:a	3	:train	0
:g1	:a	18	:train	0
:g1	:a	15	:train	0
:g3	:a	10	:test	0
:g1	:a	12	:test	0
:g3	:b	24	:test	0
:g1	:a	9	:test	0
:g3	:a	13	:test	0
...	...	...	...	...
:g1	:a	2	:test	0
:g3	:a	7	:test	0
:g2	:b	20	:test	0
:g3	:a	16	:test	0
:g2	:a	14	:test	0
:g3	:a	17	:test	0
:g1	:a	5	:test	0
:g2	:a	11	:test	0
:g2	:a	8	:test	0
:g1	:b	23	:test	0
:g1	:a	0	:test	0

you can split to more than two subdatasets with holdout

```
(tc/split for-splitting :holdout {:ratio [0.1 0.2 0.3 0.15 0.25]})
```

_unnamed, (splitted) [25 5]:

:group	:partition	:id	:*split* − *name*| :split-id	
:g2	:a	11	:train	0
:g1	:a	1	:train	0
:g1	:a	12	:test	0
:g3	:a	7	:test	0
:g3	:a	13	:test	0
:g1	:a	3	:test	0
:g1	:a	15	:test	0
:g3	:a	16	:split-2	0
:g1	:a	2	:split-2	0
:g1	:b	23	:split-2	0
...	...	...	...	...
:g3	:a	17	:split-3	0
:g3	:b	24	:split-3	0
:g1	:a	0	:split-3	0
:g1	:a	18	:split-4	0
:g2	:a	8	:split-4	0
:g2	:b	20	:split-4	0
:g1	:a	5	:split-4	0
:g3	:a	19	:split-4	0
:g3	:a	10	:split-4	0
:g1	:a	6	:split-4	0
:g2	:a	4	:split-4	0

you can use also proportions with custom names

```
(tc/split for-splitting :holdout {:ratio [5 3 11 2]
                                  :split-names ["small" "smaller" "big" "the rest"]})
```

_unnamed, (splitted) [25 5]:

:group	:partition	:id	:*split* − *name*| :split-id	
:g3	:a	10	small	0
:g3	:a	16	small	0
:g2	:b	20	small	0
:g3	:a	7	small	0
:g3	:a	13	small	0
:g1	:a	3	smaller	0
:g2	:a	8	smaller	0
:g3	:a	17	smaller	0
:g1	:a	5	big	0
:g1	:a	6	big	0
...	...	...	...	...
:g1	:a	12	big	0
:g1	:a	18	big	0
:g2	:b	22	big	0
:g1	:a	9	big	0
:g1	:a	2	big	0
:g1	:a	1	big	0
:g2	:a	4	big	0
:g1	:a	0	the rest	0
:g2	:a	11	the rest	0

:group	:partition	:id	:*split* − *name*| :split-id	
:g3	:a	19	the rest	0
:g3	:b	24	the rest	0

Holdouts With ratios from 5% to 95% of the dataset with step 1.5 generates 15 splits with ascending rows in train dataset.

```
(-> (tc/split for-splitting :holdouts {:steps [0.05 0.95 1.5]
                                       :shuffle? false})
    (tc/group-by [:$split-id :$split-name]))
```

_unnamed [30 3]:

:group-id	:name	:data
0	{:*split* − *name* : *train*, :split-id 0}	Group: {:*split* − *name* : *train*, :split-id 0} [1 5]:
1	{:*split* − *name* : *test*, :split-id 0}	Group: {:*split* − *name* : *test*, :split-id 0} [24 5]:
2	{:*split* − *name* : *train*, :split-id 1}	Group: {:*split* − *name* : *train*, :split-id 1} [2 5]:
3	{:*split* − *name* : *test*, :split-id 1}	Group: {:*split* − *name* : *test*, :split-id 1} [23 5]:
4	{:*split* − *name* : *train*, :split-id 2}	Group: {:*split* − *name* : *train*, :split-id 2} [4 5]:
5	{:*split* − *name* : *test*, :split-id 2}	Group: {:*split* − *name* : *test*, :split-id 2} [21 5]:
6	{:*split* − *name* : *train*, :split-id 3}	Group: {:*split* − *name* : *train*, :split-id 3} [5 5]:
7	{:*split* − *name* : *test*, :split-id 3}	Group: {:*split* − *name* : *test*, :split-id 3} [20 5]:
8	{:*split* − *name* : *train*, :split-id 4}	Group: {:*split* − *name* : *train*, :split-id 4} [7 5]:
9	{:*split* − *name* : *test*, :split-id 4}	Group: {:*split* − *name* : *test*, :split-id 4} [18 5]:
...	...	...
19	{:*split* − *name* : *test*, :split-id 9}	Group: {:*split* − *name* : *test*, :split-id 9} [11 5]:
20	{:*split* − *name* : *train*, :split-id 10}	Group: {:*split* − *name* : *train*, :split-id 10} [16 5]:
21	{:*split* − *name* : *test*, :split-id 10}	Group: {:*split* − *name* : *test*, :split-id 10} [9 5]:
22	{:*split* − *name* : *train*, :split-id 11}	Group: {:*split* − *name* : *train*, :split-id 11} [17 5]:
23	{:*split* − *name* : *test*, :split-id 11}	Group: {:*split* − *name* : *test*, :split-id 11} [8 5]:
24	{:*split* − *name* : *train*, :split-id 12}	Group: {:*split* − *name* : *train*, :split-id 12} [19 5]:
25	{:*split* − *name* : *test*, :split-id 12}	Group: {:*split* − *name* : *test*, :split-id 12} [6 5]:
26	{:*split* − *name* : *train*, :split-id 13}	Group: {:*split* − *name* : *train*, :split-id 13} [20 5]:
27	{:*split* − *name* : *test*, :split-id 13}	Group: {:*split* − *name* : *test*, :split-id 13} [5 5]:
28	{:*split* − *name* : *train*, :split-id 14}	Group: {:*split* − *name* : *train*, :split-id 14} [22 5]:
29	{:*split* − *name* : *test*, :split-id 14}	Group: {:*split* − *name* : *test*, :split-id 14} [3 5]:

```
(-> for-splitting
    (tc/split :loo)
    (tc/head 30))
```

Leave One Out _unnamed, (splitted) [30 5]:

:group	:partition	:id	:*split* − *name*| :split-id	
:g3	:a	13	:train	0
:g1	:a	9	:train	0
:g3	:a	17	:train	0
:g1	:a	12	:train	0
:g1	:a	5	:train	0
:g1	:b	23	:train	0

:group	:partition	:id	:*split* − *name*| :split-id	
:g1	:a	6	:train	0
:g1	:a	3	:train	0
:g3	:a	19	:train	0
:g2	:a	4	:train	0
:g1	:a	18	:train	0
:g2	:a	8	:train	0
:g2	:a	14	:train	0
:g3	:a	10	:train	0
:g3	:b	24	:train	0
:g3	:a	16	:train	0
:g1	:a	2	:train	0
:g1	:a	15	:train	0
:g1	:a	1	:train	0
:g2	:b	20	:train	0
:g2	:b	22	:train	0
:g1	:a	0	:train	0
:g1	:b	21	:train	0
:g2	:a	11	:train	0
:g3	:a	7	:test	0
:g3	:a	7	:train	1
:g1	:a	9	:train	1
:g3	:a	17	:train	1
:g1	:a	12	:train	1
:g1	:a	5	:train	1

```
(-> for-splitting
    (tc/split :loo)
    (tc/row-count))
```

625

```
(-> for-splitting
    (tc/group-by :group)
    (tc/split :bootstrap {:partition-selector :partition :seed 11 :ratio 0.8}))
```

Grouped dataset with partitioning _unnamed [3 3]:

:group-id	:name	:data
0	:g1	Group: :g1, (splitted) [15 5]:
1	:g2	Group: :g2, (splitted) [8 5]:
2	:g3	Group: :g3, (splitted) [8 5]:

Split as a sequence To get a sequence of pairs, use `split->seq` function

```
(-> for-splitting
    (tc/split->seq :kfold {:partition-selector :partition})
    (first))
```

{:train Group: 0 [20 3]:

:group	:partition	:id
:g1	:a	5
:g1	:a	2
:g2	:a	4
:g1	:a	0
:g2	:a	14
:g3	:a	19
:g1	:a	18
:g2	:a	11
:g1	:a	9
:g3	:a	13
:g3	:a	16
:g1	:a	1
:g3	:a	7
:g3	:a	10
:g1	:a	6
:g1	:a	3
:g1	:b	23
:g2	:b	22
:g2	:b	20
:g3	:b	24

, :test Group: 0 [5 3]:

:group	:partition	:id
:g3	:a	17
:g1	:a	12
:g1	:a	15
:g2	:a	8
:g1	:b	21

}

```
(-> for-splitting
    (tc/group-by :group)
    (tc/split->seq :bootstrap {:partition-selector :partition :seed 11 :ratio 0.8 :repeats 2})
    (first))
```

[:g1 ({:train Group: 0 [10 3]:

:group	:partition	:id
:g1	:a	3
:g1	:a	9
:g1	:a	9
:g1	:a	3
:g1	:a	15
:g1	:a	5
:g1	:a	2
:g1	:a	9
:g1	:b	21

:group	:partition	:id
:g1	:b	23

, :test Group: 0 [5 3]:

:group	:partition	:id
:g1	:a	0
:g1	:a	12
:g1	:a	1
:g1	:a	18
:g1	:a	6

} {:train Group: 1 [10 3]:

:group	:partition	:id
:g1	:a	0
:g1	:a	9
:g1	:a	9
:g1	:a	1
:g1	:a	6
:g1	:a	0
:g1	:a	15
:g1	:a	18
:g1	:b	23
:g1	:b	23

, :test Group: 1 [5 3]:

:group	:partition	:id
:g1	:a	12
:g1	:a	5
:g1	:a	3
:g1	:a	2
:g1	:b	21

})]

Pipeline

`tablecloth.pipeline` exports special versions of API which create functions operating only on dataset. This creates the possibility to chain operations and compose them easily.

There are two ways to create pipelines:

- functional, as a composition of functions
- declarative, separating task declarations and concrete parametrization.

Pipeline operations are prepared to work with metamorph library. That means that result of the pipeline is wrapped into a map and dataset is stored under `:metamorph/data` key.

> **Warning: Duplicated `metamorph` pipeline functions are removed from `tablecloth.pipeline` namespace.**

Functions

This API doesn't provide any statistical, numerical or date/time functions. Use below namespaces:

Namespace	functions
`tech.v3.datatype.functional`	primitive oprations, reducers, statistics
`tech.v3.datatype.datetime`	date/time converters and operations

Other examples

Stocks

```
(defonce stocks (tc/dataset "https://raw.githubusercontent.com/techascent/tech.ml.dataset/master/test/da
```

```
stocks
```

https://raw.githubusercontent.com/techascent/tech.ml.dataset/master/test/data/stocks.csv [560 3]:

:symbol	:date	:price
MSFT	2000-01-01	39.81
MSFT	2000-02-01	36.35
MSFT	2000-03-01	43.22
MSFT	2000-04-01	28.37
MSFT	2000-05-01	25.45
MSFT	2000-06-01	32.54
MSFT	2000-07-01	28.40
MSFT	2000-08-01	28.40
MSFT	2000-09-01	24.53
MSFT	2000-10-01	28.02
...	...	...
AAPL	2009-05-01	135.81
AAPL	2009-06-01	142.43
AAPL	2009-07-01	163.39
AAPL	2009-08-01	168.21
AAPL	2009-09-01	185.35
AAPL	2009-10-01	188.50
AAPL	2009-11-01	199.91
AAPL	2009-12-01	210.73
AAPL	2010-01-01	192.06
AAPL	2010-02-01	204.62
AAPL	2010-03-01	223.02

```
(-> stocks
    (tc/group-by (fn [row]
                   {:symbol (:symbol row)
                    :year (tech.v3.datatype.datetime/long-temporal-field :years (:date row))}))
    (tc/aggregate #(tech.v3.datatype.functional/mean (% :price)))
    (tc/order-by [:symbol :year]))
```

_unnamed [51 3]:

summary	:year	:symbol
21.74833333	2000	AAPL
10.17583333	2001	AAPL
9.40833333	2002	AAPL
9.34750000	2003	AAPL
18.72333333	2004	AAPL
48.17166667	2005	AAPL
72.04333333	2006	AAPL
133.35333333	2007	AAPL
138.48083333	2008	AAPL
150.39333333	2009	AAPL
...	...	...
29.67333333	2000	MSFT
25.34750000	2001	MSFT
21.82666667	2002	MSFT
20.93416667	2003	MSFT
22.67416667	2004	MSFT
23.84583333	2005	MSFT
24.75833333	2006	MSFT
29.28416667	2007	MSFT
25.20833333	2008	MSFT
22.87250000	2009	MSFT
28.50666667	2010	MSFT

```
(-> stocks
    (tc/group-by (juxt :symbol #(tech.v3.datatype.datetime/long-temporal-field :years (% :date))))
    (tc/aggregate #(tech.v3.datatype.functional/mean (% :price)))
    (tc/rename-columns {:$group-name-0 :symbol
                        :$group-name-1 :year}))
```

_unnamed [51 3]:

summary	:symbol	:year
29.67333333	MSFT	2000
25.34750000	MSFT	2001
21.82666667	MSFT	2002
20.93416667	MSFT	2003
22.67416667	MSFT	2004
23.84583333	MSFT	2005
24.75833333	MSFT	2006
29.28416667	MSFT	2007
25.20833333	MSFT	2008
22.87250000	MSFT	2009
...	...	...
21.74833333	AAPL	2000
10.17583333	AAPL	2001
9.40833333	AAPL	2002
9.34750000	AAPL	2003
18.72333333	AAPL	2004
48.17166667	AAPL	2005
72.04333333	AAPL	2006
133.35333333	AAPL	2007

summary	:symbol	:year
138.48083333	AAPL	2008
150.39333333	AAPL	2009
206.56666667	AAPL	2010

data.table

Below you can find comparizon between functionality of `data.table` and Clojure dataset API. I leave it without comments, please refer original document explaining details:

Introduction to `data.table`

R

```
library(data.table)
library(knitr)

flights <- fread("https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights14.csv

kable(head(flights))
```

year	month	day	dep_delay	arr_delay	carrier	origin	dest	air_time	distance	hour
2014	1	1	14	13	AA	JFK	LAX	359	2475	9
2014	1	1	-3	13	AA	JFK	LAX	363	2475	11
2014	1	1	2	9	AA	JFK	LAX	351	2475	19
2014	1	1	-8	-26	AA	LGA	PBI	157	1035	7
2014	1	1	2	1	AA	JFK	LAX	350	2475	13
2014	1	1	4	0	AA	EWR	LAX	339	2454	18

Clojure

```
(require '[tech.v3.datatype.functional :as dfn]
         '[tech.v3.datatype.argops :as aops]
         '[tech.v3.datatype :as dtype])

(defonce flights (tc/dataset "https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/f
```

```
(tc/head flights 6)
```

https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights14.csv [6 11]:

year	month	day	dep_delay	arr_delay	carrier	origin	dest	air_time	distance	hour
2014	1	1	14	13	AA	JFK	LAX	359	2475	9
2014	1	1	-3	13	AA	JFK	LAX	363	2475	11
2014	1	1	2	9	AA	JFK	LAX	351	2475	19
2014	1	1	-8	-26	AA	LGA	PBI	157	1035	7
2014	1	1	2	1	AA	JFK	LAX	350	2475	13
2014	1	1	4	0	AA	EWR	LAX	339	2454	18

Basics

Shape of loaded data R

```
dim(flights)
```

```
[1] 253316     11
```

Clojure

```
(tc/shape flights)
```

```
[253316 11]
```

What is data.table? R

```
DT = data.table(
  ID = c("b","b","b","a","a","c"),
  a = 1:6,
  b = 7:12,
  c = 13:18
)

kable(DT)
```

ID	a	b	c
b	1	7	13
b	2	8	14
b	3	9	15
a	4	10	16
a	5	11	17
c	6	12	18

```
class(DT$ID)
```

```
[1] "character"
```

Clojure

```
(def DT (tc/dataset {:ID ["b" "b" "b" "a" "a" "c"]
                     :a (range 1 7)
                     :b (range 7 13)
                     :c (range 13 19)}))
```

```
DT
```

_unnamed [6 4]:

:ID	:a	:b	:c
b	1	7	13
b	2	8	14
b	3	9	15
a	4	10	16
a	5	11	17
c	6	12	18

```
(-> :ID DT meta :datatype)
```

```
:string
```

Get all the flights with "JFK" as the origin airport in the month of June. R

```
ans <- flights[origin == "JFK" & month == 6L]
kable(head(ans))
```

year	month	day	dep_delay	arr_delay	carrier	origin	dest	air_time	distance	hour
2014	6	1	-9	-5	AA	JFK	LAX	324	2475	8
2014	6	1	-10	-13	AA	JFK	LAX	329	2475	12
2014	6	1	18	-1	AA	JFK	LAX	326	2475	7
2014	6	1	-6	-16	AA	JFK	LAX	320	2475	10
2014	6	1	-4	-45	AA	JFK	LAX	326	2475	18
2014	6	1	-6	-23	AA	JFK	LAX	329	2475	14

Clojure

```
(-> flights
    (tc/select-rows (fn [row] (and (= (get row "origin") "JFK")
                                   (= (get row "month") 6))))
    (tc/head 6))
```

https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights14.csv [6 11]:

year	month	day	dep_delay	arr_delay	carrier	origin	dest	air_time	distance	hour
2014	6	1	-9	-5	AA	JFK	LAX	324	2475	8
2014	6	1	-10	-13	AA	JFK	LAX	329	2475	12
2014	6	1	18	-1	AA	JFK	LAX	326	2475	7
2014	6	1	-6	-16	AA	JFK	LAX	320	2475	10
2014	6	1	-4	-45	AA	JFK	LAX	326	2475	18
2014	6	1	-6	-23	AA	JFK	LAX	329	2475	14

Get the first two rows from `flights`. R

```
ans <- flights[1:2]
kable(ans)
```

year	month	day	dep_delay	arr_delay	carrier	origin	dest	air_time	distance	hour
2014	1	1	14	13	AA	JFK	LAX	359	2475	9
2014	1	1	-3	13	AA	JFK	LAX	363	2475	11

Clojure

```
(tc/select-rows flights (range 2))
```

https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights14.csv [2 11]:

year	month	day	dep_delay	arr_delay	carrier	origin	dest	air_time	distance	hour
2014	1	1	14	13	AA	JFK	LAX	359	2475	9
2014	1	1	-3	13	AA	JFK	LAX	363	2475	11

Sort `flights` first by column `origin` in ascending order, and then by `dest` in descending order R

```
ans <- flights[order(origin, -dest)]
kable(head(ans))
```

year	month	day	dep_delay	arr_delay	carrier	origin	dest	air_time	distance	hour
2014	1	5	6	49	EV	EWR	XNA	195	1131	8
2014	1	6	7	13	EV	EWR	XNA	190	1131	8
2014	1	7	-6	-13	EV	EWR	XNA	179	1131	8
2014	1	8	-7	-12	EV	EWR	XNA	184	1131	8
2014	1	9	16	7	EV	EWR	XNA	181	1131	8
2014	1	13	66	66	EV	EWR	XNA	188	1131	9

Clojure

```
(-> flights
    (tc/order-by ["origin" "dest"] [:asc :desc])
    (tc/head 6))
```

https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights14.csv [6 11]:

year	month	day	dep_delay	arr_delay	carrier	origin	dest	air_time	distance	hour
2014	6	3	-6	-38	EV	EWR	XNA	154	1131	6
2014	1	20	-9	-17	EV	EWR	XNA	177	1131	8
2014	3	19	-6	10	EV	EWR	XNA	201	1131	6
2014	2	3	231	268	EV	EWR	XNA	184	1131	12
2014	4	25	-8	-32	EV	EWR	XNA	159	1131	6
2014	2	19	21	10	EV	EWR	XNA	176	1131	8

Select `arr_delay` column, but return it as a vector R

```
ans <- flights[, arr_delay]
head(ans)
```

```
[1]  13  13   9 -26   1   0
```

Clojure

```
(take 6 (flights "arr_delay"))
```

```
(13 13 9 -26 1 0)
```

Select `arr_delay` column, but return as a data.table instead R

```
ans <- flights[, list(arr_delay)]
kable(head(ans))
```

arr_delay
13
13
9
-26
1
0

Clojure

```
(-> flights
    (tc/select-columns "arr_delay")
    (tc/head 6))
```

https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights14.csv [6 1]:

arr_delay
13
13
9
-26
1
0

Select both `arr_delay` and `dep_delay` columns R

```
ans <- flights[, .(arr_delay, dep_delay)]
kable(head(ans))
```

arr_delay	dep_delay
13	14
13	-3
9	2
-26	-8
1	2
0	4

Clojure

```
(-> flights
    (tc/select-columns ["arr_delay" "dep_delay"])
    (tc/head 6))
```

https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights14.csv [6 2]:

arr_delay	dep_delay
13	14
13	-3
9	2
-26	-8
1	2
0	4

Select both `arr_delay` and `dep_delay` columns and rename them to `delay_arr` and `delay_dep` R

```
ans <- flights[, .(delay_arr = arr_delay, delay_dep = dep_delay)]
kable(head(ans))
```

delay_arr	delay_dep
13	14
13	-3
9	2
-26	-8
1	2
0	4

Clojure

```
(-> flights
    (tc/select-columns {"arr_delay" "delay_arr"
                        "dep_delay" "delay_arr"})
    (tc/head 6))
```

https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights14.csv [6 2]:

delay_arr	delay_arr
13	14
13	-3
9	2
-26	-8
1	2
0	4

How many trips have had total delay < 0? R

```
ans <- flights[, sum( (arr_delay + dep_delay) < 0 )]
ans
```

```
[1] 141814
```

Clojure

```
(->> (dfn/+ (flights "arr_delay") (flights "dep_delay"))
     (aops/argfilter #(< % 0.0))
     (dtype/ecount))
```

```
141814
```

or pure Clojure functions (much, much slower)

```
(->> (map + (flights "arr_delay") (flights "dep_delay"))
     (filter neg?)
     (count))
```

```
141814
```

Calculate the average arrival and departure delay for all flights with "JFK" as the origin airport in the month of June R

```
ans <- flights[origin == "JFK" & month == 6L,
               .(m_arr = mean(arr_delay), m_dep = mean(dep_delay))]
kable(ans)
```

m_arr	m_dep
5.839349	9.807884

Clojure

```
(-> flights
    (tc/select-rows (fn [row] (and (= (get row "origin") "JFK")
                                   (= (get row "month") 6))))
    (tc/aggregate {:m_arr #(dfn/mean (% "arr_delay"))
                   :m_dep #(dfn/mean (% "dep_delay"))}))
```

_unnamed [1 2]:

:m_arr	:m_dep
5.83934932	9.80788411

How many trips have been made in 2014 from "JFK" airport in the month of June? R

```
ans <- flights[origin == "JFK" & month == 6L, length(dest)]
ans
```

```
[1] 8422
```

or

```
ans <- flights[origin == "JFK" & month == 6L, .N]
ans
```

```
[1] 8422
```

Clojure

```
(-> flights
    (tc/select-rows (fn [row] (and (= (get row "origin") "JFK")
                                   (= (get row "month") 6))))
    (tc/row-count))
```

```
8422
```

deselect columns using - or ! R

```
ans <- flights[, !c("arr_delay", "dep_delay")]
kable(head(ans))
```

year	month	day	carrier	origin	dest	air_time	distance	hour
2014	1	1	AA	JFK	LAX	359	2475	9
2014	1	1	AA	JFK	LAX	363	2475	11
2014	1	1	AA	JFK	LAX	351	2475	19
2014	1	1	AA	LGA	PBI	157	1035	7
2014	1	1	AA	JFK	LAX	350	2475	13
2014	1	1	AA	EWR	LAX	339	2454	18

or

```
ans <- flights[, -c("arr_delay", "dep_delay")]
kable(head(ans))
```

year	month	day	carrier	origin	dest	air_time	distance	hour
2014	1	1	AA	JFK	LAX	359	2475	9
2014	1	1	AA	JFK	LAX	363	2475	11
2014	1	1	AA	JFK	LAX	351	2475	19
2014	1	1	AA	LGA	PBI	157	1035	7
2014	1	1	AA	JFK	LAX	350	2475	13
2014	1	1	AA	EWR	LAX	339	2454	18

Clojure

```
(-> flights
    (tc/select-columns (complement #{"arr_delay" "dep_delay"}))
    (tc/head 6))
```

https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights14.csv [6 9]:

year	month	day	carrier	origin	dest	air_time	distance	hour
2014	1	1	AA	JFK	LAX	359	2475	9
2014	1	1	AA	JFK	LAX	363	2475	11
2014	1	1	AA	JFK	LAX	351	2475	19
2014	1	1	AA	LGA	PBI	157	1035	7
2014	1	1	AA	JFK	LAX	350	2475	13
2014	1	1	AA	EWR	LAX	339	2454	18

Aggregations

How can we get the number of trips corresponding to each origin airport? R

```
ans <- flights[, .(.N), by = .(origin)]
kable(ans)
```

origin	N
JFK	81483
LGA	84433
EWR	87400

Clojure

```
(-> flights
    (tc/group-by ["origin"])
    (tc/aggregate {:N tc/row-count}))
```

_unnamed [3 2]:

:N	origin
81483	JFK
84433	LGA
87400	EWR

How can we calculate the number of trips for each origin airport for carrier code "AA"? R

```
ans <- flights[carrier == "AA", .N, by = origin]
kable(ans)
```

origin	N
JFK	11923
LGA	11730
EWR	2649

Clojure

```
(-> flights
    (tc/select-rows #(= (get % "carrier") "AA"))
    (tc/group-by ["origin"])
    (tc/aggregate {:N tc/row-count}))
```

_unnamed [3 2]:

:N	origin
11923	JFK
11730	LGA
2649	EWR

How can we get the total number of trips for each `origin`, `dest` pair for carrier code "AA"? R

```
ans <- flights[carrier == "AA", .N, by = .(origin, dest)]
kable(head(ans))
```

origin	dest	N
JFK	LAX	3387
LGA	PBI	245
EWR	LAX	62
JFK	MIA	1876
JFK	SEA	298
EWR	MIA	848

Clojure

```
(-> flights
    (tc/select-rows #(= (get % "carrier") "AA"))
    (tc/group-by ["origin" "dest"])
    (tc/aggregate {:N tc/row-count})
    (tc/head 6))
```

_unnamed [6 3]:

:N	origin	dest
3387	JFK	LAX
245	LGA	PBI
62	EWR	LAX
1876	JFK	MIA
298	JFK	SEA
848	EWR	MIA

How can we get the average arrival and departure delay for each `orig`,`dest` pair for each month for carrier code "AA"? R

```
ans <- flights[carrier == "AA",
        .(mean(arr_delay), mean(dep_delay)),
        by = .(origin, dest, month)]
kable(head(ans,10))
```

origin	dest	month	V1	V2
JFK	LAX	1	6.590361	14.2289157
LGA	PBI	1	-7.758621	0.3103448
EWR	LAX	1	1.366667	7.5000000
JFK	MIA	1	15.720670	18.7430168
JFK	SEA	1	14.357143	30.7500000
EWR	MIA	1	11.011236	12.1235955
JFK	SFO	1	19.252252	28.6396396
JFK	BOS	1	12.919643	15.2142857
JFK	ORD	1	31.586207	40.1724138
JFK	IAH	1	28.857143	14.2857143

Clojure

```
(-> flights
    (tc/select-rows #(= (get % "carrier") "AA"))
    (tc/group-by ["origin" "dest" "month"])
    (tc/aggregate [#(dfn/mean (% "arr_delay"))
                   #(dfn/mean (% "dep_delay"))])
    (tc/head 10))
```

_unnamed [10 5]:

month	:summary-0	:summary-1	origin	dest
1	6.59036145	14.22891566	JFK	LAX
1	-7.75862069	0.31034483	LGA	PBI
1	1.36666667	7.50000000	EWR	LAX
1	15.72067039	18.74301676	JFK	MIA
1	14.35714286	30.75000000	JFK	SEA
1	11.01123596	12.12359551	EWR	MIA
1	19.25225225	28.63963964	JFK	SFO
1	12.91964286	15.21428571	JFK	BOS
1	31.58620690	40.17241379	JFK	ORD
1	28.85714286	14.28571429	JFK	IAH

So how can we directly order by all the grouping variables? R

```
ans <- flights[carrier == "AA",
        .(mean(arr_delay), mean(dep_delay)),
        keyby = .(origin, dest, month)]
kable(head(ans,10))
```

origin	dest	month	V1	V2
EWR	DFW	1	6.427673	10.012579
EWR	DFW	2	10.536765	11.345588
EWR	DFW	3	12.865031	8.079755
EWR	DFW	4	17.792683	12.920732
EWR	DFW	5	18.487805	18.682927
EWR	DFW	6	37.005952	38.744048
EWR	DFW	7	20.250000	21.154762
EWR	DFW	8	16.936046	22.069767
EWR	DFW	9	5.865031	13.055215
EWR	DFW	10	18.813665	18.894410

Clojure

```
(-> flights
    (tc/select-rows #(= (get % "carrier") "AA"))
    (tc/group-by ["origin" "dest" "month"])
    (tc/aggregate [#(dfn/mean (% "arr_delay"))
                   #(dfn/mean (% "dep_delay"))])
    (tc/order-by ["origin" "dest" "month"])
    (tc/head 10))
```

_unnamed [10 5]:

month	:summary-0	:summary-1	origin	dest
1	6.42767296	10.01257862	EWR	DFW
2	10.53676471	11.34558824	EWR	DFW
3	12.86503067	8.07975460	EWR	DFW
4	17.79268293	12.92073171	EWR	DFW
5	18.48780488	18.68292683	EWR	DFW
6	37.00595238	38.74404762	EWR	DFW
7	20.25000000	21.15476190	EWR	DFW
8	16.93604651	22.06976744	EWR	DFW
9	5.86503067	13.05521472	EWR	DFW
10	18.81366460	18.89440994	EWR	DFW

Can `by` accept expressions as well or does it just take columns? R

```
ans <- flights[, .N, .(dep_delay>0, arr_delay>0)]
kable(ans)
```

dep_delay	arr_delay	N
TRUE	TRUE	72836
FALSE	TRUE	34583
FALSE	FALSE	119304
TRUE	FALSE	26593

Clojure

```
(-> flights
    (tc/group-by (fn [row]
                   {:dep_delay (pos? (get row "dep_delay"))
                    :arr_delay (pos? (get row "arr_delay"))}))
    (tc/aggregate {:N tc/row-count}))
```

_unnamed [4 3]:

:arr_delay	:dep_delay	:N
true	true	72836
true	false	34583
false	false	119304
false	true	26593

Do we have to compute `mean()` for each column individually? R

```
kable(DT)
```

ID	a	b	c
b	1	7	13
b	2	8	14
b	3	9	15
a	4	10	16
a	5	11	17

ID	a	b	c
c	6	12	18

```
DT[, print(.SD), by = ID]
```

```
   a b  c
1: 1 7 13
2: 2 8 14
3: 3 9 15
   a  b  c
1: 4 10 16
2: 5 11 17
   a  b  c
1: 6 12 18
```

```
Empty data.table (0 rows and 1 cols): ID
```

```
kable(DT[, lapply(.SD, mean), by = ID])
```

ID	a	b	c
b	2.0	8.0	14.0
a	4.5	10.5	16.5
c	6.0	12.0	18.0

Clojure

```
DT

(tc/group-by DT :ID {:result-type :as-map})
```

_unnamed [6 4]:

:ID	:a	:b	:c
b	1	7	13
b	2	8	14
b	3	9	15
a	4	10	16
a	5	11	17
c	6	12	18

{“b” Group: b [3 4]:

:ID	:a	:b	:c
b	1	7	13
b	2	8	14
b	3	9	15

, “a” Group: a [2 4]:

:ID	:a	:b	:c
a	4	10	16
a	5	11	17

, “c” Group: c [1 4]:

:ID	:a	:b	:c
c	6	12	18

}

```
(-> DT
    (tc/group-by [:ID])
    (tc/aggregate-columns (complement #{:ID}) dfn/mean))
```

_unnamed [3 4]:

:b	:c	:a	:ID
8.0	14.0	2.0	b
10.5	16.5	4.5	a
12.0	18.0	6.0	c

How can we specify just the columns we would like to compute the `mean()` on? R

```
kable(head(flights[carrier == "AA",                     ## Only on trips with carrier "AA"
                   lapply(.SD, mean),                   ## compute the mean
                   by = .(origin, dest, month),         ## for every 'origin,dest,month'
                   .SDcols = c("arr_delay", "dep_delay")])) ## for just those specified in .SDcols
```

origin	dest	month	arr_delay	dep_delay
JFK	LAX	1	6.590361	14.2289157
LGA	PBI	1	-7.758621	0.3103448
EWR	LAX	1	1.366667	7.5000000
JFK	MIA	1	15.720670	18.7430168
JFK	SEA	1	14.357143	30.7500000
EWR	MIA	1	11.011236	12.1235955

Clojure

```
(-> flights
    (tc/select-rows #(= (get % "carrier") "AA"))
    (tc/group-by ["origin" "dest" "month"])
    (tc/aggregate-columns ["arr_delay" "dep_delay"] dfn/mean)
    (tc/head 6))
```

_unnamed [6 5]:

arr_delay	month	origin	dep_delay	dest
6.59036145	1	JFK	14.22891566	LAX
-7.75862069	1	LGA	0.31034483	PBI
1.36666667	1	EWR	7.50000000	LAX
15.72067039	1	JFK	18.74301676	MIA
14.35714286	1	JFK	30.75000000	SEA
11.01123596	1	EWR	12.12359551	MIA

How can we return the first two rows for each month? R

```
ans <- flights[, head(.SD, 2), by = month]
kable(head(ans))
```

month	year	day	dep_delay	arr_delay	carrier	origin	dest	air_time	distance	hour
1	2014	1	14	13	AA	JFK	LAX	359	2475	9
1	2014	1	-3	13	AA	JFK	LAX	363	2475	11
2	2014	1	-1	1	AA	JFK	LAX	358	2475	8
2	2014	1	-5	3	AA	JFK	LAX	358	2475	11
3	2014	1	-11	36	AA	JFK	LAX	375	2475	8
3	2014	1	-3	14	AA	JFK	LAX	368	2475	11

Clojure

```
(-> flights
    (tc/group-by ["month"])
    (tc/head 2) ;; head applied on each group
    (tc/ungroup)
    (tc/head 6))
```

_unnamed [6 11]:

year	month	day	dep_delay	arr_delay	carrier	origin	dest	air_time	distance	hour
2014	1	1	14	13	AA	JFK	LAX	359	2475	9
2014	1	1	-3	13	AA	JFK	LAX	363	2475	11
2014	2	1	-1	1	AA	JFK	LAX	358	2475	8
2014	2	1	-5	3	AA	JFK	LAX	358	2475	11
2014	3	1	-11	36	AA	JFK	LAX	375	2475	8
2014	3	1	-3	14	AA	JFK	LAX	368	2475	11

How can we concatenate columns a and b for each group in ID? R

```
kable(DT[, .(val = c(a,b)), by = ID])
```

ID	val
b	1
b	2
b	3
b	7
b	8

ID	val
b	9
a	4
a	5
a	10
a	11
c	6
c	12

Clojure

```
(-> DT
    (tc/pivot->longer [:a :b] {:value-column-name :val})
    (tc/drop-columns [:$column :c]))
```

_unnamed [12 2]:

:ID	:val
b	1
b	2
b	3
a	4
a	5
c	6
b	7
b	8
b	9
a	10
a	11
c	12

What if we would like to have all the values of column `a` and `b` concatenated, but returned as a list column? R

```
kable(DT[, .(val = list(c(a,b))), by = ID])
```

ID	val
b	1, 2, 3, 7, 8, 9
a	4, 5, 10, 11
c	6, 12

Clojure

```
(-> DT
    (tc/pivot->longer [:a :b] {:value-column-name :val})
    (tc/drop-columns [:$column :c])
    (tc/fold-by :ID))
```

_unnamed [3 2]:

:ID	:val
b	[1 2 3 7 8 9]
a	[4 5 10 11]
c	[6 12]

API tour

Below snippets are taken from A data.table and dplyr tour written by Atrebas (permission granted).

I keep structure and subtitles but I skip `data.table` and `dplyr` examples.

Example data

```
(def DS (tc/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"]))}))
```

```
(tc/dataset? DS)
(class DS)
```

```
true
tech.v3.dataset.impl.dataset.Dataset
```

```
DS
```

_unnamed [9 4]:

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

Basic Operations

Filter rows Filter rows using indices

```
(tc/select-rows DS [2 3])
```

_unnamed [2 4]:

:V1	:V2	:V3	:V4
1	3	1.5	C
2	4	0.5	A

Discard rows using negative indices

In Clojure API we have separate function for that: `drop-rows`.

```
(tc/drop-rows DS (range 2 7))
```

_unnamed [4 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	2	1.0	B
2	8	1.0	B
1	9	1.5	C

Filter rows using a logical expression

```
(tc/select-rows DS (comp #(> % 5) :V2))
```

_unnamed [4 4]:

:V1	:V2	:V3	:V4
2	6	1.5	C
1	7	0.5	A
2	8	1.0	B
1	9	1.5	C

```
(tc/select-rows DS (comp #{"A" "C"} :V4))
```

_unnamed [6 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
1	3	1.5	C
2	4	0.5	A
2	6	1.5	C
1	7	0.5	A
1	9	1.5	C

Filter rows using multiple conditions

```
(tc/select-rows DS #(and (= (:V1 %) 1)
                         (= (:V4 %) "A")))
```

_unnamed [2 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
1	7	0.5	A

Filter unique rows

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

_unnamed [9 4]:

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

```
(tc/unique-by DS [:V1 :V4])
```

_unnamed [6 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	2	1.0	B
1	3	1.5	C
2	4	0.5	A
1	5	1.0	B
2	6	1.5	C

Discard rows with missing values

```
(tc/drop-missing DS)
```

_unnamed [9 4]:

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

Other filters

```
(tc/random DS 3) ;; 3 random rows
```

_unnamed [3 4]:

:V1	:V2	:V3	:V4
1	1	0.5	A
2	6	1.5	C
1	1	0.5	A

```
(tc/random DS (/ (tc/row-count DS) 2)) ;; fraction of random rows
```

_unnamed [5 4]:

:V1	:V2	:V3	:V4
2	2	1.0	B
1	1	0.5	A
2	2	1.0	B
1	1	0.5	A
2	2	1.0	B

```
(tc/by-rank DS :V1 zero?) ;; take top n entries
```

_unnamed [4 4]:

:V1	:V2	:V3	:V4
2	2	1.0	B
2	4	0.5	A
2	6	1.5	C
2	8	1.0	B

Convenience functions

```
(tc/select-rows DS (comp (partial re-matches #"^B") str :V4))
```

_unnamed [3 4]:

:V1	:V2	:V3	:V4
2	2	1.0	B
1	5	1.0	B
2	8	1.0	B

```
(tc/select-rows DS (comp #(<= 3 % 5) :V2))
```

_unnamed [3 4]:

:V1	:V2	:V3	:V4
1	3	1.5	C
2	4	0.5	A
1	5	1.0	B

```
(tc/select-rows DS (comp #(< 3 % 5) :V2))
```

_unnamed [1 4]:

:V1	:V2	:V3	:V4
2	4	0.5	A

```
(tc/select-rows DS (comp #(<= 3 % 5) :V2))
```

_unnamed [3 4]:

:V1	:V2	:V3	:V4
1	3	1.5	C
2	4	0.5	A
1	5	1.0	B

Last example skipped.

Sort rows Sort rows by column

```
(tc/order-by DS :V3)
```

_unnamed [9 4]:

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

Sort rows in decreasing order

```
(tc/order-by DS :V3 :desc)
```

_unnamed [9 4]:

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

:V1	:V2	:V3	:V4
2	4	0.5	A
1	1	0.5	A

Sort rows based on several columns

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

_unnamed [9 4]:

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

Select columns Select one column using an index (not recommended)

```
(nth (tc/columns DS :as-seq) 2) ;; as column (iterable)
```

```
#tech.v3.dataset.column<float64>[9]
:V3
[0.5000, 1.000, 1.500, 0.5000, 1.000, 1.500, 0.5000, 1.000, 1.500]
```

```
(tc/dataset [(nth (tc/columns DS :as-seq) 2)])
```

_unnamed [9 1]:

:V3
0.5
1.0
1.5
0.5
1.0
1.5
0.5
1.0
1.5

Select one column using column name

```
(tc/select-columns DS :V2) ;; as dataset
```

_unnamed [9 1]:

:V2
1
2
3
4
5
6
7
8
9

```
(tc/select-columns DS [:V2]) ;; as dataset
```

_unnamed [9 1]:

:V2
1
2
3
4
5
6
7
8
9

```
(DS :V2) ;; as column (iterable)
```

```
#tech.v3.dataset.column<int64>[9]
:V2
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Select several columns

```
(tc/select-columns DS [:V2 :V3 :V4])
```

_unnamed [9 3]:

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

Exclude columns

```
(tc/select-columns DS (complement #{:V2 :V3 :V4}))
```

_unnamed [9 1]:

:V1
1
2
1
2
1
2
1
2
1

```
(tc/drop-columns DS [:V2 :V3 :V4])
```

_unnamed [9 1]:

:V1
1
2
1
2
1
2
1
2
1

Other seletions

```
(->> (range 1 3)
     (map (comp keyword (partial format "V%d")))
     (tc/select-columns DS))
```

_unnamed [9 2]:

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

```
(tc/reorder-columns DS :V4)
```

_unnamed [9 4]:

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

```
(tc/select-columns DS #(clojure.string/starts-with? (name %) "V"))
```

_unnamed [9 4]:

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

```
(tc/select-columns DS #(clojure.string/ends-with? (name %) "3"))
```

_unnamed [9 1]:

:V3
0.5
1.0
1.5
0.5
1.0
1.5
0.5
1.0
1.5

```
(tc/select-columns DS #"..2") ;; regex converts to string using `str` function
```

_unnamed [9 1]:

:V2
1
2
3
4
5
6
7
8
9

```
(tc/select-columns DS #{:V1 "X"})
```

_unnamed [9 1]:

:V1
1
2
1
2
1
2
1
2
1

```
(tc/select-columns DS #(not (clojure.string/starts-with? (name %) "V2")))
```

_unnamed [9 3]:

:V1	:V3	:V4
1	0.5	A
2	1.0	B
1	1.5	C
2	0.5	A
1	1.0	B
2	1.5	C
1	0.5	A
2	1.0	B
1	1.5	C

Summarise data Summarise one column

```
(reduce + (DS :V1)) ;; using pure Clojure, as value
```

```
13
```

```
(tc/aggregate-columns DS :V1 dfn/sum) ;; as dataset
```

_unnamed [1 1]:

:V1
13.0

```
(tc/aggregate DS {:sumV1 #(dfn/sum (% :V1))})
```

_unnamed [1 1]:

:sumV1
13.0

Summarize several columns

```
(tc/aggregate DS [#(dfn/sum (% :V1))
                  #(dfn/standard-deviation (% :V3))])
```

_unnamed [1 2]:

:summary-0	:summary-1
13.0	0.4330127

```
(tc/aggregate-columns DS [:V1 :V3] [dfn/sum
                                    dfn/standard-deviation])
```

_unnamed [1 2]:

:V1	:V3
13.0	0.4330127

Summarise several columns and assign column names

```
(tc/aggregate DS {:sumv1 #(dfn/sum (% :V1))
                  :sdv3 #(dfn/standard-deviation (% :V3))})
```

_unnamed [1 2]:

:sumv1	:sdv3
13.0	0.4330127

Summarise a subset of rows

```
(-> DS
    (tc/select-rows (range 4))
    (tc/aggregate-columns :V1 dfn/sum))
```

_unnamed [1 1]:

:V1
6.0

```
(-> DS
    (tc/first)
    (tc/select-columns :V3)) ;; select first row from `:V3` column
```

Additional helpers _unnamed [1 1]:

:V3
0.5

```
(-> DS
    (tc/last)
    (tc/select-columns :V3)) ;; select last row from `:V3` column
```

_unnamed [1 1]:

:V3
1.5

```
(-> DS
    (tc/select-rows 4)
    (tc/select-columns :V3)) ;; select forth row from `:V3` column
```

_unnamed [1 1]:

:V3
1.0

```
(-> DS
    (tc/select :V3 4)) ;; select forth row from `:V3` column
```

_unnamed [1 1]:

:V3
1.0

```
(-> DS
    (tc/unique-by :V4)
    (tc/aggregate tc/row-count)) ;; number of unique rows in `:V4` column, as dataset
```

_unnamed [1 1]:

summary
3

```
(-> DS
    (tc/unique-by :V4)
    (tc/row-count)) ;; number of unique rows in `:V4` column, as value
```

```
3
```

```
(-> DS
    (tc/unique-by)
    (tc/row-count)) ;; number of unique rows in dataset, as value
```

```
9
```

Add/update/delete columns Modify a column

```
(tc/map-columns DS :V1 [:V1] #(dfn/pow % 2))
```

_unnamed [9 4]:

:V1	:V2	:V3	:V4
1.0	1	0.5	A
4.0	2	1.0	B
1.0	3	1.5	C
4.0	4	0.5	A
1.0	5	1.0	B
4.0	6	1.5	C
1.0	7	0.5	A
4.0	8	1.0	B
1.0	9	1.5	C

```
(def DS (tc/add-column DS :V1 (dfn/pow (DS :V1) 2)))
```

```
DS
```

_unnamed [9 4]:

:V1	:V2	:V3	:V4
1.0	1	0.5	A
4.0	2	1.0	B
1.0	3	1.5	C
4.0	4	0.5	A
1.0	5	1.0	B
4.0	6	1.5	C
1.0	7	0.5	A
4.0	8	1.0	B
1.0	9	1.5	C

Add one column

```
(tc/map-columns DS :v5 [:V1] dfn/log)
```

_unnamed [9 5]:

:V1	:V2	:V3	:V4	:v5
1.0	1	0.5	A	0.00000000
4.0	2	1.0	B	1.38629436
1.0	3	1.5	C	0.00000000
4.0	4	0.5	A	1.38629436
1.0	5	1.0	B	0.00000000
4.0	6	1.5	C	1.38629436
1.0	7	0.5	A	0.00000000
4.0	8	1.0	B	1.38629436
1.0	9	1.5	C	0.00000000

```
(def DS (tc/add-column DS :v5 (dfn/log (DS :V1))))
```

```
DS
```

_unnamed [9 5]:

:V1	:V2	:V3	:V4	:v5
1.0	1	0.5	A	0.00000000
4.0	2	1.0	B	1.38629436
1.0	3	1.5	C	0.00000000
4.0	4	0.5	A	1.38629436
1.0	5	1.0	B	0.00000000
4.0	6	1.5	C	1.38629436
1.0	7	0.5	A	0.00000000
4.0	8	1.0	B	1.38629436
1.0	9	1.5	C	0.00000000

Add several columns

```
(def DS (tc/add-columns DS {:v6 (dfn/sqrt (DS :V1))
                            :v7 "X"}))
```

```
DS
```

_unnamed [9 7]:

:V1	:V2	:V3	:V4	:v5	:v6	:v7
1.0	1	0.5	A	0.00000000	1.0	X
4.0	2	1.0	B	1.38629436	2.0	X
1.0	3	1.5	C	0.00000000	1.0	X
4.0	4	0.5	A	1.38629436	2.0	X
1.0	5	1.0	B	0.00000000	1.0	X
4.0	6	1.5	C	1.38629436	2.0	X
1.0	7	0.5	A	0.00000000	1.0	X
4.0	8	1.0	B	1.38629436	2.0	X

:V1	:V2	:V3	:V4	:v5	:v6	:v7
1.0	9	1.5	C	0.00000000	1.0	X

Create one column and remove the others

```
(tc/dataset {:v8 (dfn/+ (DS :V3) 1)})
```

_unnamed [9 1]:

:v8
1.5
2.0
2.5
1.5
2.0
2.5
1.5
2.0
2.5

Remove one column

```
(def DS (tc/drop-columns DS :v5))
```

```
DS
```

_unnamed [9 6]:

:V1	:V2	:V3	:V4	:v6	:v7
1.0	1	0.5	A	1.0	X
4.0	2	1.0	B	2.0	X
1.0	3	1.5	C	1.0	X
4.0	4	0.5	A	2.0	X
1.0	5	1.0	B	1.0	X
4.0	6	1.5	C	2.0	X
1.0	7	0.5	A	1.0	X
4.0	8	1.0	B	2.0	X
1.0	9	1.5	C	1.0	X

Remove several columns

```
(def DS (tc/drop-columns DS [:v6 :v7]))
```

```
DS
```

_unnamed [9 4]:

:V1	:V2	:V3	:V4
1.0	1	0.5	A
4.0	2	1.0	B
1.0	3	1.5	C
4.0	4	0.5	A
1.0	5	1.0	B
4.0	6	1.5	C
1.0	7	0.5	A
4.0	8	1.0	B
1.0	9	1.5	C

Remove columns using a vector of colnames

We use set here.

```
(def DS (tc/select-columns DS (complement #{:V3})))
```

```
DS
```

_unnamed [9 3]:

:V1	:V2	:V4
1.0	1	A
4.0	2	B
1.0	3	C
4.0	4	A
1.0	5	B
4.0	6	C
1.0	7	A
4.0	8	B
1.0	9	C

Replace values for rows matching a condition

```
(def DS (tc/map-columns DS :V2 [:V2] #(if (< % 4.0) 0.0 %)))
```

```
DS
```

_unnamed [9 3]:

:V1	:V2	:V4
1.0	0.0	A
4.0	0.0	B
1.0	0.0	C
4.0	4.0	A
1.0	5.0	B
4.0	6.0	C
1.0	7.0	A
4.0	8.0	B
1.0	9.0	C

by By group

```
(-> DS
    (tc/group-by [:V4])
    (tc/aggregate {:sumV2 #(dfn/sum (% :V2))}))
```

_unnamed [3 2]:

:V4	:sumV2
A	11.0
B	13.0
C	15.0

By several groups

```
(-> DS
    (tc/group-by [:V4 :V1])
    (tc/aggregate {:sumV2 #(dfn/sum (% :V2))}))
```

_unnamed [6 3]:

:V4	:V1	:sumV2
A	1.0	7.0
B	4.0	8.0
C	1.0	9.0
A	4.0	4.0
B	1.0	5.0
C	4.0	6.0

Calling function in by

```
(-> DS
    (tc/group-by (fn [row]
                   (clojure.string/lower-case (:V4 row))))
    (tc/aggregate {:sumV1 #(dfn/sum (% :V1))}))
```

_unnamed [3 2]:

:sumV1	:$group-name
6.0	a
9.0	b
6.0	c

Assigning column name in by

```
(-> DS
    (tc/group-by (fn [row]
                   {:abc (clojure.string/lower-case (:V4 row))}))
    (tc/aggregate {:sumV1 #(dfn/sum (% :V1))}))
```

_unnamed [3 2]:

:abc	:sumV1
a	6.0
b	9.0
c	6.0

```
(-> DS
    (tc/group-by (fn [row]
                   (clojure.string/lower-case (:V4 row))))
    (tc/aggregate {:sumV1 #(dfn/sum (% :V1))} {:add-group-as-column :abc}))
```

_unnamed [3 2]:

:abc	:sumV1
a	6.0
b	9.0
c	6.0

Using a condition in by

```
(-> DS
    (tc/group-by #(= (:V4 %) "A"))
    (tc/aggregate #(dfn/sum (% :V1))))
```

_unnamed [2 2]:

summary	:$group-name
6.0	true
15.0	false

By on a subset of rows

```
(-> DS
    (tc/select-rows (range 5))
    (tc/group-by :V4)
    (tc/aggregate {:sumV1 #(dfn/sum (% :V1))}))
```

_unnamed [3 2]:

:sumV1	:$group-name
5.0	A
5.0	B
1.0	C

Count number of observations for each group

```
(-> DS
    (tc/group-by :V4)
    (tc/aggregate tc/row-count))
```

_unnamed [3 2]:

summary	:$group-name
3	A
3	B
3	C

Add a column with number of observations for each group

```
(-> DS
    (tc/group-by [:V1])
    (tc/add-column :n tc/row-count)
    (tc/ungroup))
```

_unnamed [9 4]:

:V1	:V2	:V4	:n
1.0	0.0	A	5
1.0	0.0	C	5
1.0	5.0	B	5
1.0	7.0	A	5
1.0	9.0	C	5
4.0	0.0	B	4
4.0	4.0	A	4
4.0	6.0	C	4
4.0	8.0	B	4

Retrieve the first/last/nth observation for each group

```
(-> DS
    (tc/group-by [:V4])
    (tc/aggregate-columns :V2 first))
```

_unnamed [3 2]:

:V2	:V4
0.0	A
0.0	B
0.0	C

```
(-> DS
    (tc/group-by [:V4])
    (tc/aggregate-columns :V2 last))
```

_unnamed [3 2]:

:V2	:V4
7.0	A
8.0	B
9.0	C

```
(-> DS
    (tc/group-by [:V4])
    (tc/aggregate-columns :V2 #(nth % 1)))
```

_unnamed [3 2]:

:V2	:V4
4.0	A
5.0	B
6.0	C

Going further

Advanced columns manipulation Summarise all the columns

```
;; custom max function which works on every type
(tc/aggregate-columns DS :all (fn [col] (first (sort #(compare %2 %1) col))))
```

_unnamed [1 3]:

:V1	:V2	:V4
4.0	9.0	C

Summarise several columns

```
(tc/aggregate-columns DS [:V1 :V2] dfn/mean)
```

_unnamed [1 2]:

:V1	:V2
2.33333333	4.33333333

Summarise several columns by group

```
(-> DS
    (tc/group-by [:V4])
    (tc/aggregate-columns [:V1 :V2] dfn/mean))
```

_unnamed [3 3]:

:V2	:V4	:V1
3.66666667	A	2.0
4.33333333	B	3.0
5.00000000	C	2.0

Summarise with more than one function by group

```
(-> DS
    (tc/group-by [:V4])
    (tc/aggregate-columns [:V1 :V2] (fn [col]
                                      {:sum (dfn/sum col)
                                       :mean (dfn/mean col)})))
```

_unnamed [3 5]:

:V2-sum	:V1-mean	:V4	:V2-mean	:V1-sum
11.0	2.0	A	3.66666667	6.0
13.0	3.0	B	4.33333333	9.0
15.0	2.0	C	5.00000000	6.0

Summarise using a condition

```
(-> DS
    (tc/select-columns :type/numerical)
    (tc/aggregate-columns :all dfn/mean))
```

_unnamed [1 2]:

:V1	:V2
2.33333333	4.33333333

Modify all the columns

```
(tc/update-columns DS :all reverse)
```

_unnamed [9 3]:

:V1	:V2	:V4
1.0	9.0	C
4.0	8.0	B
1.0	7.0	A
4.0	6.0	C
1.0	5.0	B
4.0	4.0	A
1.0	0.0	C
4.0	0.0	B
1.0	0.0	A

Modify several columns (dropping the others)

```
(-> DS
    (tc/select-columns [:V1 :V2])
    (tc/update-columns :all dfn/sqrt))
```

_unnamed [9 2]:

:V1	:V2
1.0	0.00000000
2.0	0.00000000
1.0	0.00000000
2.0	2.00000000
1.0	2.23606798
2.0	2.44948974
1.0	2.64575131
2.0	2.82842712
1.0	3.00000000

```
(-> DS
    (tc/select-columns (complement #{:V4}))
    (tc/update-columns :all dfn/exp))
```

_unnamed [9 2]:

:V1	:V2
2.71828183	1.00000000
54.59815003	1.00000000
2.71828183	1.00000000
54.59815003	54.59815003
2.71828183	148.41315910
54.59815003	403.42879349
2.71828183	1096.63315843
54.59815003	2980.95798704
2.71828183	8103.08392758

Modify several columns (keeping the others)

```
(def DS (tc/update-columns DS [:V1 :V2] dfn/sqrt))
```

```
DS
```

_unnamed [9 3]:

:V1	:V2	:V4
1.0	0.00000000	A
2.0	0.00000000	B
1.0	0.00000000	C
2.0	2.00000000	A
1.0	2.23606798	B
2.0	2.44948974	C

:V1	:V2	:V4
1.0	2.64575131	A
2.0	2.82842712	B
1.0	3.00000000	C

```
(def DS (tc/update-columns DS (complement #{:V4}) #(dfn/pow % 2)))
```

```
DS
```

_unnamed [9 3]:

:V1	:V2	:V4
1.0	0.0	A
4.0	0.0	B
1.0	0.0	C
4.0	4.0	A
1.0	5.0	B
4.0	6.0	C
1.0	7.0	A
4.0	8.0	B
1.0	9.0	C

Modify columns using a condition (dropping the others)

```
(-> DS
    (tc/select-columns :type/numerical)
    (tc/update-columns :all #(dfn/- % 1)))
```

_unnamed [9 2]:

:V1	:V2
0.0	-1.0
3.0	-1.0
0.0	-1.0
3.0	3.0
0.0	4.0
3.0	5.0
0.0	6.0
3.0	7.0
0.0	8.0

Modify columns using a condition (keeping the others)

```
(def DS (tc/convert-types DS :type/numerical :int32))
```

```
DS
```

_unnamed [9 3]:

:V1	:V2	:V4
1	0	A
4	0	B
1	0	C
4	4	A
1	5	B
4	5	C
1	7	A
4	8	B
1	9	C

Use a complex expression

```
(-> DS
    (tc/group-by [:V4])
    (tc/head 2)
    (tc/add-column :V2 "X")
    (tc/ungroup))
```

_unnamed [6 3]:

:V1	:V2	:V4
1	X	A
4	X	A
4	X	B
1	X	B
1	X	C
4	X	C

Use multiple expressions

```
(tc/dataset (let [x (dfn/+ (DS :V1) (dfn/sum (DS :V2)))]
              (println (seq (DS :V1)))
              (println (tc/info (tc/select-columns DS :V1)))
              {:A (range 1 (inc (tc/row-count DS)))
               :B x}))
```

(1 4 1 4 1 4 1 4 1) _unnamed: descriptive-stats [1 11]:

:col-name	:datatype	:n-valid	:n-missing	:min	:mean	:max	:standard-deviation	:skew	:first	:last
:V1	:int32	9	0	1.0	2.33333333	4.0	1.58113883	0.27105237	1	1

_unnamed [9 2]:

:A	:B
1	39.0
2	42.0

:A	:B
3	39.0
4	42.0
5	39.0
6	42.0
7	39.0
8	42.0
9	39.0

Chain expressions Expression chaining using >

```
(-> DS
    (tc/group-by [:V4])
    (tc/aggregate {:V1sum #(dfn/sum (% :V1))})
    (tc/select-rows #(>= (:V1sum %) 5)))
```

_unnamed [3 2]:

:V4	:V1sum
A	6.0
B	9.0
C	6.0

```
(-> DS
    (tc/group-by [:V4])
    (tc/aggregate {:V1sum #(dfn/sum (% :V1))})
    (tc/order-by :V1sum :desc))
```

_unnamed [3 2]:

:V4	:V1sum
B	9.0
A	6.0
C	6.0

Indexing and Keys Set the key/index (order)

```
(def DS (tc/order-by DS :V4))
```

```
DS
```

_unnamed [9 3]:

:V1	:V2	:V4
1	0	A
4	4	A
1	7	A
4	0	B
1	5	B
4	8	B
1	0	C

:V1	:V2	:V4
4	5	C
1	9	C

Select the matching rows

```
(tc/select-rows DS #(= (:V4 %) "A"))
```

_unnamed [3 3]:

:V1	:V2	:V4
1	0	A
4	4	A
1	7	A

```
(tc/select-rows DS (comp #{"A" "C"} :V4))
```

_unnamed [6 3]:

:V1	:V2	:V4
1	0	A
4	4	A
1	7	A
1	0	C
4	5	C
1	9	C

Select the first matching row

```
(-> DS
    (tc/select-rows #(= (:V4 %) "B"))
    (tc/first))
```

_unnamed [1 3]:

:V1	:V2	:V4
4	0	B

```
(-> DS
    (tc/unique-by :V4)
    (tc/select-rows (comp #{"B" "C"} :V4)))
```

_unnamed [2 3]:

:V1	:V2	:V4
4	0	B
1	0	C

Select the last matching row

```
(-> DS
    (tc/select-rows #(= (:V4 %) "A"))
    (tc/last))
```

_unnamed [1 3]:

:V1	:V2	:V4
1	7	A

Nomatch argument

```
(tc/select-rows DS (comp #{"A" "D"} :V4))
```

_unnamed [3 3]:

:V1	:V2	:V4
1	0	A
4	4	A
1	7	A

Apply a function on the matching rows

```
(-> DS
    (tc/select-rows (comp #{"A" "C"} :V4))
    (tc/aggregate-columns :V1 (fn [col]
                                {:sum (dfn/sum col)})))
```

_unnamed [1 1]:

:V1-sum
12.0

Modify values for matching rows

```
(def DS (-> DS
            (tc/map-columns :V1 [:V1 :V4] #(if (= %2 "A") 0 %1))
            (tc/order-by :V4)))
```

```
DS
```

_unnamed [9 3]:

:V1	:V2	:V4
0	0	A
0	4	A
0	7	A

:V1	:V2	:V4
4	0	B
1	5	B
4	8	B
1	0	C
4	5	C
1	9	C

Use keys in by

```
(-> DS
    (tc/select-rows (comp (complement #{"B"}) :V4))
    (tc/group-by [:V4])
    (tc/aggregate-columns :V1 dfn/sum))
```

_unnamed [2 2]:

:V4	:V1
A	0.0
C	6.0

Set keys/indices for multiple columns (ordered)

```
(tc/order-by DS [:V4 :V1])
```

_unnamed [9 3]:

:V1	:V2	:V4
0	0	A
0	4	A
0	7	A
1	5	B
4	0	B
4	8	B
1	0	C
1	9	C
4	5	C

Subset using multiple keys/indices

```
(-> DS
    (tc/select-rows #(and (= (:V1 %) 1)
                          (= (:V4 %) "C"))))
```

_unnamed [2 3]:

:V1	:V2	:V4
1	0	C
1	9	C

```
(-> DS
    (tc/select-rows #(and (= (:V1 %) 1)
                          (#{"B" "C"} (:V4 %)))))
```

_unnamed [3 3]:

:V1	:V2	:V4
1	5	B
1	0	C
1	9	C

```
(-> DS
    (tc/select-rows #(and (= (:V1 %) 1)
                          (#{"B" "C"} (:V4 %))) {:result-type :as-indexes}))
```

```
(4 6 8)
```

set*() modifications Replace values

There is no mutating operations `tech.ml.dataset` or easy way to set value.

```
(def DS (tc/update-columns DS :V2 #(map-indexed (fn [idx v]
                                                  (if (zero? idx) 3 v)) %)))
```

```
DS
```

_unnamed [9 3]:

:V1	:V2	:V4
0	3	A
0	4	A
0	7	A
4	0	B
1	5	B
4	8	B
1	0	C
4	5	C
1	9	C

Reorder rows

```
(def DS (tc/order-by DS [:V4 :V1] [:asc :desc]))
```

```
DS
```

_unnamed [9 3]:

:V1	:V2	:V4
0	3	A
0	4	A
0	7	A
4	0	B
4	8	B
1	5	B
4	5	C
1	0	C
1	9	C

Modify colnames

```
(def DS (tc/rename-columns DS {:V2 "v2"}))
```

```
DS
```

_unnamed [9 3]:

:V1	v2	:V4
0	3	A
0	4	A
0	7	A
4	0	B
4	8	B
1	5	B
4	5	C
1	0	C
1	9	C

```
(def DS (tc/rename-columns DS {"v2" :V2})) ;; revert back
```

Reorder columns

```
(def DS (tc/reorder-columns DS :V4 :V1 :V2))
```

```
DS
```

_unnamed [9 3]:

:V4	:V1	:V2
A	0	3
A	0	4
A	0	7
B	4	0
B	4	8
B	1	5
C	4	5
C	1	0
C	1	9

Advanced use of by Select first/last/... row by group

```
(-> DS
    (tc/group-by :V4)
    (tc/first)
    (tc/ungroup))
```

_unnamed [3 3]:

:V4	:V1	:V2
A	0	3
B	4	0
C	4	5

```
(-> DS
    (tc/group-by :V4)
    (tc/select-rows [0 2])
    (tc/ungroup))
```

_unnamed [6 3]:

:V4	:V1	:V2
A	0	3
A	0	7
B	4	0
B	1	5
C	4	5
C	1	9

```
(-> DS
    (tc/group-by :V4)
    (tc/tail 2)
    (tc/ungroup))
```

_unnamed [6 3]:

:V4	:V1	:V2
A	0	4
A	0	7
B	4	8
B	1	5
C	1	0
C	1	9

Select rows using a nested query

```
(-> DS
    (tc/group-by :V4)
    (tc/order-by :V2)
    (tc/first)
    (tc/ungroup))
```

_unnamed [3 3]:

:V4	:V1	:V2
A	0	3
B	4	0
C	1	0

Add a group counter column

```
(-> DS
    (tc/group-by [:V4 :V1])
    (tc/ungroup {:add-group-id-as-column :Grp}))
```

_unnamed [9 4]:

:Grp	:V4	:V1	:V2
0	A	0	3
0	A	0	4
0	A	0	7
1	B	4	0
1	B	4	8
2	B	1	5
3	C	4	5
4	C	1	0
4	C	1	9

Get row number of first (and last) observation by group

```
(-> DS
    (tc/add-column :row-id (range))
    (tc/select-columns [:V4 :row-id])
    (tc/group-by :V4)
    (tc/ungroup))
```

_unnamed [9 2]:

:V4	:row-id
A	0
A	1
A	2
B	3
B	4
B	5
C	6
C	7
C	8

```
(-> DS
    (tc/add-column :row-id (range))
    (tc/select-columns [:V4 :row-id])
```

```
    (tc/group-by :V4)
    (tc/first)
    (tc/ungroup))
```

_unnamed [3 2]:

:V4	:row-id
A	0
B	3
C	6

```
(-> DS
    (tc/add-column :row-id (range))
    (tc/select-columns [:V4 :row-id])
    (tc/group-by :V4)
    (tc/select-rows [0 2])
    (tc/ungroup))
```

_unnamed [6 2]:

:V4	:row-id
A	0
A	2
B	3
B	5
C	6
C	8

Handle list-columns by group

```
(-> DS
    (tc/select-columns [:V1 :V4])
    (tc/fold-by :V4))
```

_unnamed [3 2]:

:V4	:V1
A	[0 0 0]
B	[4 4 1]
C	[4 1 1]

```
(-> DS
    (tc/group-by :V4)
    (tc/unmark-group))
```

_unnamed [3 3]:

:group-id	:name	:data
0	A	Group: A [3 3]:
1	B	Group: B [3 3]:
2	C	Group: C [3 3]:

Grouping sets (multiple by at once)

Not available.

Miscellaneous

Read / Write data Write data to a csv file

```
(tc/write! DS "DF.csv")
```

```
nil
```

Write data to a tab-delimited file

```
(tc/write! DS "DF.txt" {:separator \tab})
```

```
nil
```

or

```
(tc/write! DS "DF.tsv")
```

```
nil
```

Read a csv / tab-delimited file

```
(tc/dataset "DF.csv" {:key-fn keyword})
```

DF.csv [9 3]:

:V4	:V1	:V2
A	0	3
A	0	4
A	0	7
B	4	0
B	4	8
B	1	5
C	4	5
C	1	0
C	1	9

```
(tc/dataset "DF.txt" {:key-fn keyword})
```

DF.txt [9 1]:

:V4	V1	V2
A	0	3
A	0	4
A	0	7
B	4	0
B	4	8
B	1	5
C	4	5
C	1	0
C	1	9

```
(tc/dataset "DF.tsv" {:key-fn keyword})
```

DF.tsv [9 3]:

:V4	:V1	:V2
A	0	3
A	0	4
A	0	7
B	4	0
B	4	8
B	1	5
C	4	5
C	1	0
C	1	9

Read a csv file selecting / droping columns

```
(tc/dataset "DF.csv" {:key-fn keyword
                      :column-whitelist ["V1" "V4"]})
```

DF.csv [9 2]:

:V4	:V1
A	0
A	0
A	0
B	4
B	4
B	1
C	4
C	1
C	1

```
(tc/dataset "DF.csv" {:key-fn keyword
                      :column-blacklist ["V4"]})
```

DF.csv [9 2]:

:V1	:V2
0	3
0	4
0	7
4	0
4	8
1	5
4	5
1	0
1	9

Read and rbind several files

```
(apply tc/concat (map tc/dataset ["DF.csv" "DF.csv"]))
```

DF.csv [18 3]:

V4	V1	V2
A	0	3
A	0	4
A	0	7
B	4	0
B	4	8
B	1	5
C	4	5
C	1	0
C	1	9
A	0	3
A	0	4
A	0	7
B	4	0
B	4	8
B	1	5
C	4	5
C	1	0
C	1	9

Reshape data Melt data (from wide to long)

```
(def mDS (tc/pivot->longer DS [:V1 :V2] {:target-columns :variable
                                         :value-column-name :value}))
```

```
mDS
```

_unnamed [18 3]:

:V4	:variable	:value
A	:V1	0
A	:V1	0
A	:V1	0
B	:V1	4

:V4	:variable	:value
B	:V1	4
B	:V1	1
C	:V1	4
C	:V1	1
C	:V1	1
A	:V2	3
A	:V2	4
A	:V2	7
B	:V2	0
B	:V2	8
B	:V2	5
C	:V2	5
C	:V2	0
C	:V2	9

Cast data (from long to wide)

```
(-> mDS
    (tc/pivot->wider :variable :value {:fold-fn vec})
    (tc/update-columns ["V1" "V2"] (partial map count)))
```

_unnamed [3 3]:

:V4	V1	V2
A	3	3
B	3	3
C	3	3

```
(-> mDS
    (tc/pivot->wider :variable :value {:fold-fn vec})
    (tc/update-columns ["V1" "V2"] (partial map dfn/sum)))
```

_unnamed [3 3]:

:V4	V1	V2
A	0.0	14.0
B	9.0	13.0
C	6.0	14.0

```
(-> mDS
    (tc/map-columns :value #(> % 5))
    (tc/pivot->wider :value :variable {:fold-fn vec})
    (tc/update-columns ["true" "false"] (partial map #(if (sequential? %) (count %) 1))))
```

_unnamed [3 3]:

:V4	false	true
A	5	1
B	5	1
C	5	1

Split

```
(tc/group-by DS :V4 {:result-type :as-map})
```

{“A” Group: A [3 3]:

:V4	:V1	:V2
A	0	3
A	0	4
A	0	7

, “B” Group: B [3 3]:

:V4	:V1	:V2
B	4	0
B	4	8
B	1	5

, “C” Group: C [3 3]:

:V4	:V1	:V2
C	4	5
C	1	0
C	1	9

}

Split and transpose a vector/column

```
(-> {:a ["A:a" "B:b" "C:c"]}
    (tc/dataset)
    (tc/separate-column :a [:V1 :V2] ":"))
```

_unnamed [3 2]:

:V1	:V2
A	a
B	b
C	c

Other Skipped

```
(def x (tc/dataset {"Id" ["A" "B" "C" "C"]
                    "X1" [1 3 5 7]
                    "XY" ["x2" "x4" "x6" "x8"]}))
(def y (tc/dataset {"Id" ["A" "B" "B" "D"]
                    "Y1" [1 3 5 7]
                    "XY" ["y1" "y3" "y5" "y7"]}))
```

```
x y
```

Join/Bind data sets _unnamed [4 3]:

Id	X1	XY
A	1	x2
B	3	x4
C	5	x6
C	7	x8

_unnamed [4 3]:

Id	Y1	XY
A	1	y1
B	3	y3
B	5	y5
D	7	y7

Join Join matching rows from y to x

```
(tc/left-join x y "Id")
```

left-outer-join [5 6]:

Id	X1	XY	right.Id	Y1	right.XY
A	1	x2	A	1	y1
B	3	x4	B	3	y3
B	3	x4	B	5	y5
C	5	x6			
C	7	x8			

Join matching rows from x to y

```
(tc/right-join x y "Id")
```

right-outer-join [4 6]:

Id	X1	XY	right.Id	Y1	right.XY
A	1	x2	A	1	y1
B	3	x4	B	3	y3

Id	X1	XY	right.Id	Y1	right.XY
B	3	x4	B	5	y5
			D	7	y7

Join matching rows from both x and y

```
(tc/inner-join x y "Id")
```

inner-join [3 5]:

Id	X1	XY	Y1	right.XY
A	1	x2	1	y1
B	3	x4	3	y3
B	3	x4	5	y5

Join keeping all the rows

```
(tc/full-join x y "Id")
```

full-join [6 6]:

Id	X1	XY	right.Id	Y1	right.XY
A	1	x2	A	1	y1
B	3	x4	B	3	y3
B	3	x4	B	5	y5
C	5	x6			
C	7	x8			
			D	7	y7

Return rows from x matching y

```
(tc/semi-join x y "Id")
```

semi-join [2 3]:

Id	X1	XY
A	1	x2
B	3	x4

Return rows from x not matching y

```
(tc/anti-join x y "Id")
```

anti-join [2 3]:

Id	X1	XY
C	5	x6
C	7	x8

More joins Select columns while joining

```
(tc/right-join (tc/select-columns x ["Id" "X1"])
               (tc/select-columns y ["Id" "XY"])
               "Id")
```

right-outer-join [4 4]:

Id	X1	right.Id	XY
A	1	A	y1
B	3	B	y3
B	3	B	y5
		D	y7

```
(tc/right-join (tc/select-columns x ["Id" "XY"])
               (tc/select-columns y ["Id" "XY"])
               "Id")
```

right-outer-join [4 4]:

Id	XY	right.Id	right.XY
A	x2	A	y1
B	x4	B	y3
B	x4	B	y5
		D	y7

Aggregate columns while joining

```
(-> y
    (tc/group-by ["Id"])
    (tc/aggregate {"sumY1" #(dfn/sum (% "Y1"))})
    (tc/right-join x "Id")
    (tc/add-column "X1Y1" (fn [ds] (dfn/* (ds "sumY1")
                                          (ds "X1"))))
    (tc/select-columns ["right.Id" "X1Y1"]))
```

right-outer-join [4 2]:

right.Id	X1Y1
A	1.0
B	24.0
C	
C	

Update columns while joining

```
(-> x
    (tc/select-columns ["Id" "X1"])
    (tc/map-columns "SqX1" "X1" (fn [x] (* x x)))
    (tc/right-join y "Id")
    (tc/drop-columns ["X1" "Id"]))
```

right-outer-join [4 4]:

SqX1	right.Id	Y1	XY
1	A	1	y1
9	B	3	y3
9	B	5	y5
	D	7	y7

Adds a list column with rows from y matching x (nest-join)

```
(-> (tc/left-join x y "Id")
    (tc/drop-columns ["right.Id"])
    (tc/fold-by (tc/column-names x)))
```

_unnamed [4 5]:

XY	X1	Id	right.XY	Y1
x2	1	A	[“y1”]	[1]
x4	3	B	[“y3” “y5”]	[3 5]
x6	5	C	[]	[]
x8	7	C	[]	[]

Some joins are skipped

Cross join

```
(def cjds (tc/dataset {:V1 [[2 1 1]]
                       :V2 [[3 2]]}))
```

```
cjds
```

_unnamed [1 2]:

:V1	:V2
[2 1 1]	[3 2]

```
(reduce #(tc/unroll %1 %2) cjds (tc/column-names cjds))
```

_unnamed [6 2]:

:V1	:V2
2	3
2	2
1	3
1	2
1	3
1	2

```
(-> (reduce #(tc/unroll %1 %2) cjds (tc/column-names cjds))
    (tc/unique-by))
```

_unnamed [4 2]:

:V1	:V2
2	3
2	2
1	3
1	2

```
(def x (tc/dataset {:V1 [1 2 3]}))
(def y (tc/dataset {:V1 [4 5 6]}))
(def z (tc/dataset {:V1 [7 8 9]
                    :V2 [0 0 0]}))
```

```
x y z
```

Bind _unnamed [3 1]:

:V1
1
2
3

_unnamed [3 1]:

:V1
4
5
6

_unnamed [3 2]:

:V1	:V2
7	0
8	0

:V1	:V2
9	0

Bind rows

```
(tc/bind x y)
```

_unnamed [6 1]:

:V1
1
2
3
4
5
6

```
(tc/bind x z)
```

_unnamed [6 2]:

:V1	:V2
1	
2	
3	
7	0
8	0
9	0

Bind rows using a list

```
(->> [x y]
     (map-indexed #(tc/add-column %2 :id (repeat %1)))
     (apply tc/bind))
```

_unnamed [6 2]:

:V1	:id
1	0
2	0
3	0
4	1
5	1
6	1

Bind columns

```
(tc/append x y)
```

_unnamed [3 2]:

:V1	:V1
1	4
2	5
3	6

```
(def x (tc/dataset {:V1 [1 2 2 3 3]}))
(def y (tc/dataset {:V1 [2 2 3 4 4]}))
```

```
x y
```

Set operations _unnamed [5 1]:

:V1
1
2
2
3
3

_unnamed [5 1]:

:V1
2
2
3
4
4

Intersection

```
(tc/intersect x y)
```

intersection [2 1]:

:V1
2
3

Difference

```
(tc/difference x y)
```

difference [1 1]:

:V1
1

Union

```
(tc/union x y)
```

union [4 1]:

:V1
1
2
3
4

```
(tc/concat x y)
```

_unnamed [10 1]:

:V1
1
2
2
3
3
2
2
3
4
4

Equality not implemented