Table Objects (taboo)

Version 0.1

Alex Baker

https://github.com/ambaker1/taboo

September 14, 2023

Abstract

The "taboo" package implements an object-oriented tabular datatype in Tcl, using the object variable framework provided by the vutil package. This datatype is suitable for row-oriented two-dimensional data, and efficiently handles sparse tables.

Tabular Data Structure

This package provides an object-oriented tabular datatype in Tcl, building upon the type system framework provided by the vutil package.

The string representation of this datatype is a five element dictionary with keys of "keyname", "fieldname", "keys", "fields", and "data". The property "keyname" describes what the table keys represent, and the property "fieldname" describes what the table fields represent (this is not typically present in raw table formats such as CSV). The property "keys" is an ordered list of all the row names of the table, and the property "fields" is an ordered list of all the field names of the table. The property "data" stores the table values in an unordered, sparse, nested dictionary, with the first level data keys corresponding to the table keys, and the second level data keys corresponding to the table fields. The conceptual layout of the five properties of a table is illustrated in the figure below.

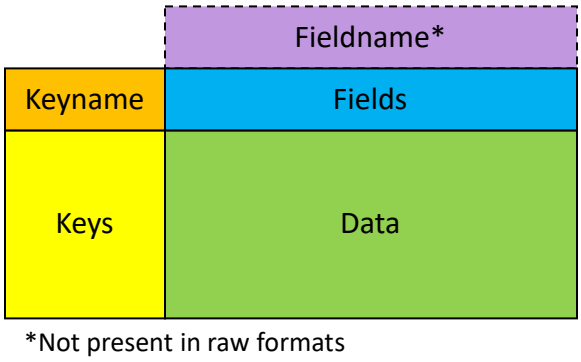

Figure 1: The five properties of a table

Creating Table Objects

This package provides the "table" type class, using the type system provided by the "vutil" package. So, tables can be created with the "vutil" command *new*, or directly with the *::taboo::table* class. Once created, table objects act as commands with an ensemble of subcommands, or methods.

```
::taboo::table new $refName <$arg...>
```

$refName	Reference name to tie object to.
$arg...	Arguments to pass to *define* method.

Example 1: Creating a table object with "vutil" *new* command

Code:

```
new table T
```

Standard Methods

Because the table objects are object variables, they have the same basic methods provided by the "vutil" package. For more info on these methods, see the documentation for the "vutil" package.

```
$tableObj --> $refName
$tableObj <- $object
$tableObj = $value
$tableObj ::= $body
$tableObj info <$field>
$tableObj print <-nonewline> <$channelID>
```

$refName	Reference name to copy to.
$object	Table object.
$value	Table value to assign.
$body	Tcl script to evaluate and set as table value.
$field	Field to query (fields "height" and "width" added).
$channelID	Open channel to print to.

Note that the methods `.=` and `:=` are also available, but they are not recommended for tables. Also note that "taboo" tables are always initialized, so the "exists" field of the object variable will always be true.

Table Properties

The method *define* sets the property values of a table, filtering the data or adding keys and fields as necessary. For example, if the keys are defined to be a subset of the current fields, it will filter the data to only include the key subset. Also, if the data is defined, all existing data will be wiped, and any new keys or fields will be added.

```
$tableObj define <$properties> <$option $value ...>
```

$properties	Dictionary of properties. Mutually exclusive with option-value syntax.
$option	Property to set: “keyname”, “fieldname”, “keys”, “fields” or “data”.
$value	Value to set property to.

The remaining examples in this documentation will use the table as defined below:

Example 2: Example Table

Code:

```
set tableObj [table new]
$tableObj define data {
    1 {x 3.44 y 7.11 z 8.67}
    2 {x 4.61 y 1.81 z 7.63}
    3 {x 8.25 y 7.56 z 3.84}
    4 {x 5.20 y 6.78 z 1.11}
    5 {x 3.26 y 9.92 z 4.56}
}
```

Wiping/Clearing a Table

The method *wipe* removes all data from a table object, so that its state is the same as a fresh table. The method *clear* only removes the data and keys stored in the table, keeping the fields and other metadata.

```
$tableObj wipe
```

```
$tableObj clear
```

Table Property Query

The method *properties* simply returns a dictionary of the table properties, as defined with *$tableObj define*. Additionally, calling the table object without any arguments will return the table properties.

```
$tableObj properties <$option>
```

$option	Property to get: “keyname”, “fieldname”, “keys”, “fields” or “data”.

Example 3: Getting table properties (and trimming a table)

Code:

```
puts [$tableObj properties]; # Automatically generates keys and fields
$tableObj define keys {1 2} fields x; # Trims data
puts [$tableObj properties]
puts [$tableObj]
```

Output:

```
keyname key fieldname field keys {1 2 3 4 5} fields {x y z} data {1 {x 3.44 y 7.11 z 8.67} 2
     {x 4.61 y 1.81 z 7.63} 3 {x 8.25 y 7.56 z 3.84} 4 {x 5.20 y 6.78 z 1.11} 5 {x 3.26 y
    9.92 z 4.56}}
keyname key fieldname field keys {1 2} fields x data {1 {x 3.44} 2 {x 4.61}}
keyname key fieldname field keys {1 2} fields x data {1 {x 3.44} 2 {x 4.61}}
```

Get Keyname and Fieldname

The keyname and fieldname properties of a table can be accessed directly with their respective methods.

```
$tableObj keyname
```

```
$tableObj fieldname
```

Get Keys and Fields

The table keys and fields are ordered lists of the row and column names of the table. They can be queried with the methods *keys* and *fields*, respectively. In addition to just returning the lists of keys and fields, a pattern can be specified in the same style as the Tcl *string match* command.

```
$tableObj keys <$i> <$pattern>
```

```
$tableObj fields <$j> <$pattern>
```

$pattern	String matching pattern. Default "*" returns all.
$i/$j	Index arguments, using "ndlist" index notation. Default ":" for all keys/fields. Does not allow "*" index notation.

Example 4: Accessing table keys

Code:

```
puts [$tableObj keys]
puts [$tableObj keys 0:end-1]
```

Output:

```
1 2 3 4 5
1 2 3 4
```

Get Single Key/Field at Row/Column Index

The table key or field corresponding with a row or column index, parsed with *::ndlist::Index2Integer*, can be queried with the methods *key* and *field*.

```
$tableObj key $i
```

```
$tableObj field $j
```

$i	Row index.
$j	Column index.

Get Table Data (Dictionary Form)

The method *data* returns the table data in unsorted dictionary form, where blanks are represented by missing dictionary entries.

```
$tableObj data <$key>
```

$key	Key to get row dictionary from (default returns all rows).

Example 5: Getting table data

Code:

```
puts [$tableObj data]
puts [$tableObj data 3]
```

Output:

```
1 {x 3.44 y 7.11 z 8.67} 2 {x 4.61 y 1.81 z 7.63} 3 {x 8.25 y 7.56 z 3.84} 4 {x 5.20 y 6.78
    z 1.11} 5 {x 3.26 y 9.92 z 4.56}
x 8.25 y 7.56 z 3.84
```

Get Table Data (Matrix Form)

The method *values* returns a matrix (list of rows) that represents the data in the table, where the rows correspond to the keys and the columns correspond to the fields. Missing entries are represented by blanks in the matrix unless specified otherwise.

```
$tableObj values <$filler>
```

$filler	Filler for missing values, default blank.

Example 6: Getting table values

Code:

```
puts [$tblObj values]
```

Output:

```
{3.44 7.11 8.67} {4.61 1.81 7.63} {8.25 7.56 3.84} {5.20 6.78 1.11} {3.26 9.92 4.56}
```

Get Table Dimensions

The dimensions of a table, as in the number of keys and fields, can be accessed with the method *shape*. Note that rows and columns with missing data will be counted.

```
$tableObj shape <$dim>
```

$dim	Dimension to take size along (default will return the number of rows and columns as a list) 0: Number of rows 1: Number of columns

Alternatively, the number of rows can be queried with *$tableObj height* and the number of columns can be queried with *$tableObj width*. These can also be accessed with the standard method *info*.

```
$tableObj height
```

```
$tableObj width
```

Example 7: Getting table dimensions

Code:

```
puts [$tableObj shape]
puts [$tableObj height]
puts [$tableObj width]
```

Output:

```
5 3
5
3
```

Check Existence of Table Keys/Fields

The existence of a table key, field, or combination of key/field can be queried with the method *exists*.

```
$tableObj exists key $key
$tableObj exists field $field
$tableObj exists value $key $field
```

$key	Key to check.
$field	Field to check.

Get Row/Column Indices

The row or column index of a table key or field can be queried with the methods *rid* and *cid*.

If the key or field does not exist, returns -1.

```
$tableObj rid $key
```

```
$tableObj cid $field
```

$key	Key to find.
$field	Field to find.

Example 8: Find column index of a field

Code:

```
puts [$tableObj cid y]
```

Output:

```
1
```

Table Entry and Access

Data entry and access to a table object can be done with single values with the methods *set* and *get*, entire rows with *rset* and *rget*, entire columns with *cset* and *cget*, or in matrix fashion with *mset* and *mget*. If entry keys/fields do not exist, they are added to the table. Additionally, since blank values represent missing data, setting a value to blank effectively unsets the table entry, but does not remove the key or field.

Single Value Entry and Access

The methods *set* and *get* allow for easy entry and access of single values in the table. Note that multiple field-value pairings can be used in *$tableObj set*.

```
$tableObj set $key $field $value ...
```

```
$tableObj get $key $field <$filler>
```

$key	Key of row to set/get data in/from.
$field	Field of column to set/get data in/from.
$value	Value to set.
$filler	Filler to return if value is missing. Default blank.

Example 9: Setting multiple values

Code:

```
$tableObj set 1 x 2.00 y 5.00 foo bar
puts [$tableObj data 1]
```

Output:

```
x 2.00 y 5.00 z 8.67 foo bar
```

Row Entry and Access

The methods *rset* and *rget* allow for easy row entry and access. Entry list length must match table width or be scalar. If entry list is blank, it will delete the row, but not the key.

```
$tableObj rset $key $row
```

```
$tableObj rget $key <$filler>
```

$key	Key of row to set/get.
$row	List of values (or scalar) to set.
$filler	Filler for missing values. Default blank.

Column Entry and Access

The methods *cset* and *cget* allow for easy column entry and access. Entry list length must match table height or be scalar. If entry list is blank, it will delete the column, but not the field.

```
$tableObj cset $field $column
```

```
$tableObj cget $field <$filler>
```

$field	Field of column to set/get.
$column	List of values (or scalar) to set.
$filler	Filler for missing values. Default blank.

Matrix Entry and Access

The methods *mset* and *mget* allow for easy matrix-style entry and access. Entry matrix size must match table size or be scalar.

```
$tableObj mset <$keys $fields> $matrix
```

```
$tableObj mget <$keys $fields> <$filler>
```

$keys	List of keys to set/get (default all keys).
$fields	List of keys to set/get (default all keys).
$matrix	Matrix of values (or scalar) to set.
$filler	Filler for missing values. Default blank.

Iterating Over Table Data

Table data can be looped through, row-wise, with the method *with*. Variables representing the key values and fields will be assigned their corresponding values, with blanks representing missing data. The variable representing the key (table keyname) is static, but changes made to field variables are reflected in the table. Unsetting a field variable or setting its value to blank unsets the corresponding data in the table.

```
$tableObj with $body
```

$body	Code to execute.

Example 10: Iterating over a table, accessing and modifying field values

Code:

```
set a 20.0
$tableObj add fields q
$tableObj with {
    puts [list $key $x]; # access key and field value
    set q [expr {$x*2 + $a}]; # modify field value
}
puts [$tableObj cget q]
```

Output:

```
1 3.44
2 4.61
3 8.25
4 5.20
5 3.26
26.88 29.22 36.5 30.4 26.52
```

Note: Just like in *dict with*, the key variable and field variables in *$tableObj with* persist after the loop.

Field Expressions

The method *expr* computes a list of values according to a field expression. In the same style as referring to variables with the dollar sign ($), the "at" symbol (@) is used by *$tableObj expr* to refer to field values, or row keys if the keyname is used. If any referenced fields have missing values for a table row, the corresponding result will be blank as well. The resulting list corresponds to the keys in the table.

```
$tableObj expr $fieldExpr
```

$fieldExpr	Field expression.

Editing Table Fields

Field expressions can be used to edit existing fields or add new fields in a table with the method *fedit*. If any of the referenced fields are blank, the corresponding entry will be blank as well.

```
$tableObj fedit $field $fieldExpr
```

$field	Field to set.
$fieldExpr	Field expression.

Example 11: Using field expressions

Code:

```
set a 20.0
puts [$tableObj cget x]
puts [$tableObj expr {@x*2 + $a}]
$tableObj fedit q {@x*2 + $a}
puts [$tableObj cget q]
```

Output:

```
3.44 4.61 8.25 5.20 3.26
26.88 29.22 36.5 30.4 26.52
26.88 29.22 36.5 30.4 26.52
```

Querying Keys that Match Criteria

The method *filter* returns the keys in a table that match criteria in a field expression.

```
$tableObj query $fieldExpr
```

$fieldExpr	Field expression that results in boolean value (true or false, 1 or 0).

Example 12: Getting keys that match criteria

Code:

```
puts [$tableObj query {@x > 3.0 && @y > 7.0}]
```

Output:

```
1 3 5
```

Filtering Table Based on Criteria

The method *filter* filters a table to the keys matching criteria in a field expression.

```
$tableObj filter $fieldExpr
```

$fieldExpr	Field expression that results in boolean value (true or false, 1 or 0).

Example 13: Filtering table to only include keys that match criteria

Code:

```
$tableObj filter {@x > 3.0 && @y > 7.0}
puts [$tableObj keys]
```

Output:

```
1 3 5
```

Searching a Table

Besides searching for specific field expression criteria with *$tableObj query*, keys matching criteria can be found with the method *search*. The method *search* searches a table using the Tcl *lsearch* command on the keys or field values. The default search method uses glob pattern matching, and returns matching keys. This search behavior can be changed with the various options, which are taken directly from the Tcl *lsearch* command. Therefore, while brief descriptions of the options are provided here, they are explained more in depth in the Tcl documentation, with the exception of the -inline option. The -inline option filters a table based on the search criteria.

```
$tableObj search <$option1 $option2 ...> <$field> $value
```

$option1 $option2 ...	Searching options. Valid options:
-exact	Compare strings exactly
-glob	Use glob-style pattern matching (default)
-regexp	Use regular expression matching
-sorted	Assume elements are in sorted order
-all	Get all matches, rather than the first match
-not	Negate the match(es)
-ascii	Use string comparison (default)
-dictionary	Use dictionary-style comparison
-integer	Use integer comparison
-real	Use floating-point comparison
-nocase	Search in a case-insensitive manner
-increasing	Assume increasing order (default)
-decreasing	Assume decreasing order
-bisect	Perform inexact match
-inline	Filter table instead of returning keys.
--	Signals end of options
$field	Field to search. If blank, searches keys.
$value	Value or pattern to search for

Note: If a field contains missing values, they will only be included in the search if the search options allow (e.g. blanks are included for string matching, but not for numerical matching).

Sorting a Table

The method *sort* sorts a table by keys or field values. The default sorting method is in increasing order, using string comparison. This sorting behavior can be changed with the various options, which are taken directly from the Tcl *lsort* command. Therefore, while brief descriptions of the options are provided here, they are explained more in depth in the Tcl documentation. Note: If a field contains missing values, the missing values will be last, regardless of sorting options.

```
$tableObj sort <$option1 $option2 ...> <$field1 $field2 ...>
```

$option1 $option2 ...	Sorting options. Valid options:
-ascii	Use string comparison (default)
-dictionary	Use dictionary-style comparison
-integer	Use integer comparison
-real	Use floating comparison
-increasing	Sort the list in increasing order (default)
-decreasing	Sort the list in decreasing order
-nocase	Compare in a case-insensitive manner
--	Signals end of options
$field1 $field2 ...	Fields to sort by (in order of sorting). If blank, sorts by keys.

Example 14: Searching and sorting

Code:

```
puts [$tableObj search -real x 8.25]; # returns first matching key
$tableObj sort -real x
puts [$tableObj keys]
puts [$tableObj cget x]; # table access reflects sorted keys
puts [$tableObj search -sorted -bisect -real x 5.0]
```

Output:

```
3
5 1 2 4 3
3.26 3.44 4.61 5.20 8.25
2
```

Merging Tables

Data from other tables can be merged into the table object with *$tableObj merge*. In order to merge, all the tables must have the same keyname and fieldname. If the merge is valid, the table data is combined, with later entries taking precedence. Additionally, the keys and fields are combined, such that if a key appears in any of the tables, it is in the combined table.

```
$tableObj merge $arg1 $arg2 ...
```

$arg1 $arg2 ...	Other table objects to merge into table. Does not destroy the input tables.

Example 15: Merging data from other tables

Code:

```
set newTable [table new]
$newTable set 1 x 5.00 q 6.34
$tableObj merge $newTable
$newTable destroy; # clean up
puts [$tableObj properties]
```

Output:

```
keyname key fieldname field keys {1 2 3 4 5} fields {x y z q} data {1 {x 5.00 y 7.11 z 8.67
    q 6.34} 2 {x 4.61 y 1.81 z 7.63} 3 {x 8.25 y 7.56 z 3.84} 4 {x 5.20 y 6.78 z 1.11} 5 {x
    3.26 y 9.92 z 4.56}}
```

Table Manipulation

The following methods are useful for adding, removing, and rearranging rows and columns in a table. With the exception of *$tableObj remove*, which removes corresponding data, and *$tableObj mkkey*, which may cause data loss, these methods do not add or remove data, they only modify the key and field lists.

Adding Keys/Fields

The method *add* adds keys or fields to a table, appending to the end of the key/field lists. If a key or field already exists it is ignored.

```
$tableObj add keys $arg1 $arg1 ...
$tableObj add fields $field1 $field2 ...
```

`$key1 $key2 ...`	Keys to add.
`$field1 $field2 ...`	Fields to add.

Removing Keys/Fields

The method *remove* removes keys or fields and their corresponding rows and columns from a table. If a key or field does not exist, it is ignored.

```
$tableObj remove keys $key1 $key2 ...
$tableObj remove fields $field1 $field2 ...
```

`$key1 $key2 ...`	Keys to remove.
`$field1 $field2 ...`	Fields to remove.

Cleaning a Table

Keys and fields with no data are removed with the method *clean*.

```
$tableObj clean
```

Inserting Keys/Fields

The method *insert* inserts keys or fields at a specific row or column index. Input keys or fields must be unique and must not already exist.

```
$tableObj insert keys $rid $key1 $key2 ...
$tableObj insert fields $cid $field1 $field2 ...
```

`$rid`	Row index to insert keys at.
`$key1 $key2 ...`	Keys to remove.
`$cid`	Column index to insert fields at.
`$field1 $field2 ...`	Fields to remove.

Renaming Keys/Fields

The method *rename* renames keys or fields. Old keys and fields must exist. Duplicates are not allowed in old and new key/field lists.

```
$tableObj rename keys $oldKeys $newKeys
$tableObj rename fields $oldFields $newFields
```

`$oldKeys`	Keys to rename. Must exist.
`$newKeys`	New key names. Must be same length as $oldKeys.
`$oldFields`	Fields to rename. Must exist.
`$newFields`	New field names. Must be same length as $oldFields.

Making a Field the Key of a Table

The method *mkkey* makes a field the key of a table, and makes the key a field. If a field is empty for some keys, those keys will be lost. Additionally, if field values repeat, only the last entry for that field value will be included. This method is intended to be used with a field that is full and unique, and if the keyname matches a field name, this command will return an error.

```
$tableObj mkkey $field
```

`$field`	Field to swap with key.

Swapping Rows/Columns

Existing rows and columns can be swapped with the methods *rswap* and *cswap*.

```
$tableObj rswap $key1 $key2
```

```
$tableObj cswap $field1 $field2
```

`$key1 $key2 ...`	Keys to swap.
`$field1 $field2 ...`	Fields to swap.

Moving Rows/Columns

Existing rows and columns can be moved with the methods *rmove* and *cmove*.

```
$tableObj rmove $key $rid
```

```
$tableObj cmove $field $cid
```

`$key`	Key of row to move.
`$rid`	Row index to move to.
`$field`	Field of row to move.
`$cid`	Column index to move to.

Transposing a Table

The method *transpose* transposes the table, making the keys the fields and the fields the keys.

```
$tableObj transpose
```

Example 16: Transposing a table

Code:

```
$tableObj transpose
puts [$tableObj properties]
```

Output:

```
keyname field fieldname key keys {x y z} fields {1 2 3 4 5} data {x {1 3.44 2 4.61 3 8.25 4
    5.20 5 3.26} y {1 7.11 2 1.81 3 7.56 4 6.78 5 9.92} z {1 8.67 2 7.63 3 3.84 4 1.11 5
    4.56}}
```

Command Index