

# speed tables - a high-performance, memory-resident database

# 1

This chapter gives an overview of speed tables and describes the sort of applications they are useful for.

Speed tables provides an interface for defining *tables* containing zero or more *rows*, with each row containing one or more *fields*. The speed table compiler reads the table definition and generates C code to create and manage corresponding structures, currently producing a set of C access routines and a C language extension for Tcl to create, access and manipulate those tables. It then compiles the extension, links it as a shared library, and makes it loadable on demand via Tcl's "package require" mechanism.

Speed tables are well-suited for applications for which this table/row/field abstraction is useful, with row counts from the dozens to the tens of millions, for which the performance requirements for access or update frequency exceed those of the available SQL database, and the application does not require "no transaction loss" behavior in the event of a crash.

## Example Application

Speed Tables is used as a high-speed cache that front-ends a SQL database for a website generating millions of customized page views per day using commodity hardware.

In contrast to ad-hoc tables implemented with some combination of *arrays*, *lists*, *upvar*, *namespaces*, or even using *dicts*, Speed tables' memory footprint is far smaller and performance far higher when many rows are present.

Speed tables support tab-separated reading and writing to files and TCP/IP sockets, and has a direct C interface to PostgreSQL. Examples are provided for importing SQL query results into a speed table as well as copying from a speed table to a database table. Speed tables' search function provides a number of powerful capabilities including results sorting, setting offsets and limits, specifying match expressions, and counting.

# Contents

Representing Complex Data Structures in Tcl	4
Speed Table Data Types	7
indexed	8
nonnull	8
default	8
length	8
unique	8
Example Speed Table Definition	9
Where Stuff Is Built	13
Methods for Manipulating Speed Tables	14
set	14
store	15
makekey	15
key	15
fields	15
field	16
get	16
array_get	16
array_get_with_nulls	16
exists	17
delete	17
count	18
batch	18
search	18
search+	24
foreach	24
incr	24

type	25
import_postgres_result	25
fieldtype	26
needs_quoting	26
names	26
reset	26
destroy	27
share	27
getprop	27
attach	27
statistics	27
write_tabsep	28
read_tabsep	28
index	29
Special Notes On How To Make Searches Go Fast	31
Client-Server Speed Tables	32
Shared Memory Speed Tables	37
\$ctable create name master list	38
\$ctable create name reader list	38
\$ctable attach pid	39
\$ctable getprop	39
\$ctable getprop name ...	39
share command [\$ctable share] ?options?	39
\$ctable share command ?options?	39
stapi - Speed Table API	42
stDisplay - Display Functions for the WWW	51
C Code Generated and C Routines Created	54
Interfacing with Speed Tables From C	62
Troubleshooting	63
Examples	65
Copy Speed Table Search Results to a Tab-Separated File	65
Load a Speed Table using read_tabsep	65
Using Copy In For Super Fast Speed Table-to-PostgreSQL Transfers	65

# Representing Complex Data Structures in Tcl

# 2

Representing Complex Data Structures in Tcl

This chapter describes common approaches taken to represent complex data structures in Tcl, their costs and tradeoffs, and begins to describe some of Speed Tables' capabilities.

Tcl is not well-known for its ability to represent complex data structures. Yes, it has *lists* and *associative arrays* and, in Tcl 8.5, *dicts*. Yes, object-oriented extensions such as *Incr Tcl* provide ways to plug objects together to represent fairly complex data structures and yes, the *BLT toolkit*, among others, has provided certain more efficient ways to represent data (a vector data type, for instance) than available by default and, yes, you can abuse *upvar* and *namespaces* as part of expressing the structure of, and methods of access for, your data.

There are, however, three typical problems with this approach:

1. It is memory-inefficient.

Tables implemented using Tcl objects use an order of magnitude more memory than native C.

For example, an integer, stored as a Tcl object, has the integer value and all the overhead of a Tcl object, 24 bytes minimum, routinely more, and often way more. When constructing Tcl lists, there is an overhead to making those lists, and the list structures themselves consume memory, sometimes a surprising amount as Tcl tries to avoid allocating memory on the fly by often allocating more than you need, and sometimes much more than you need.<sup>1</sup>

Another drawback of Tcl arrays is that they store the field names (keys) along with each value, which is inherently necessary given their design but is yet another example of the inefficiency of this approach.

2. It is computationally inefficient.

---

<sup>1</sup> It is common to see ten or twenty times the space consumed by the data itself used up by the Tcl objects, lists, arrays, etc, used to hold them. Even on a modern machine, using 20 gigabytes of memory to store a gigabyte of data is at a minimum kind of gross and, at worst, renders the solution unusable.)

Constructing, managing and manipulating complicated structures out of lists, arrays, etc, is quite processor-intensive when compared to, for instance, a hand-coded C-based approach exploiting pointers, C structs, and the like.

3. It yields code that is clumsy and obtuse.

## Hackish construction of complex data structures sucks

Using a combination of *upvar* and *namespaces* and *lists* and *arrays* to represent a complex structure yields relatively opaque and inflexible ways of expressing and manipulating that structure, twisting the code and typically replicating little pieces of weird structure access drivel strewn throughout the application, making the code hard to follow, teach, fix, enhance, and hand off.

**Speed tables** reads a structure definition and emits C code to create and manipulate tables of rows of that structure. We generate a full-fledged Tcl C extension that manages rows of fields as native C structs and emit subroutines for manipulating those rows in an efficient manner.

Memory efficiency is high because we have low per-row storage overhead beyond the size of the struct itself, and fields are stored in native formats such as short integer, integer, float, double, bit, etc.

Computational efficiency is high because we are reasonably clever about storing and fetching those values, particularly when populating from lines of tab-separated data as well as PostgreSQL database query results, inserting into them by reading rows from a Tcl channel containing tab-separated data, writing them tab-separated, locating them, updating them, and counting them, as well as importing and exporting by other means.

Speed tables avoids executing Tcl code on a per row basis when a lot of rows need to be looked at. In particular when bulk inserting and bulk processing via search, Tcl essentially configures an execution engine that can operate on millions of rows of data without the Tcl interpreter's per-row involvement except, perhaps, for example, executing scripted code only on the few rows that match your search criteria.

Speed tables also maintains a "null value" bit per field, unless told not to, and provide an out-of-band way to distinguish between null values and non-null values, as is present in SQL databases... providing a ready bridge between those databases and speed tables.

## Example Application

Speed tables is used as the realtime database for a monitoring system that polls millions of devices every few minutes. Device status and performance data is kept in speed tables. Information about the status of de-

vices is continually “swept” to the SQL database at a sustainable rate. The loss of even a sizable number of scan results in the event of a crash is not a serious problem, as within a few minutes of starting up, the system will have obtained fresh data by newly polling the devices.

Speed tables supports defining skip list-based indexes on one or more fields in a row, providing multi-hundred-fold speed improvements for many searches.

Fields that are not declared to be indexable do not have any code generated to check for the existence of indexes, etc, when they are changed, one of a number of optimizations performed to make speed tables fast.

# Speed Table Data Types

Speed Table Data Types

This chapter explains the various data types that can be used to create fields in a speed table.

The following data types are available<sup>2</sup>:

- *boolean* - a single 0/1 bit
- *varstring* - a variable-length string
- *fixedstring* - a fixed-length string
- *short* - a short integer
- *int* - a machine native integer
- *long* - a machine native long
- *wide* - a 64-bit wide integer (Tcl Wide)
- *float* - a floating point number
- *double* - a double-precision floating point number
- *char* - a single character (deprecation likely)
- *mac* - an ethernet MAC address
- *inet* - an internet IP address
- *tclobj* - a Tcl object... more on this powerful capability later

Fields are defined by the data type followed by the field name, for example...

double longitude

...to define a double-precision field named longitude.

---

<sup>2</sup> Additional data types can be added, although over Speed Tables' evolution it has become an increasingly complicated undertaking.

Field definitions can be followed by one or more key-value pairs that define additional attributes about the field. Supported attributes include

- *indexed*

If *indexed* is specified with a true (nonzero) value, the code generated for the speed table will include support for generating, maintaining, and using a skip list index on the field being defined.

Indexed traversal can be performed in conjunction with the speed table's search functions to accelerate searches and avoid sorts. Defaults to "indexed 0" aka the field is not generated with index support.

Indexed support is not provided for boolean fields.

- *nonnull*

If *nonnull* is specified with a true (nonzero) value, the code generated for the speed table will have code for maintaining an out-of-band null/not-null status suppressed, resulting in a substantial performance increase for fields for which out-of-band null support is not needed. Defaults to "nonnull 0" aka null values are supported.

- *default*

If *default* is specified, the following value is defined as the default value and will be set into rows that are created when the field does not have a value assigned.

There is no default default value, however if no default value is defined and the field is declared as *nonnull*, strings will default to empty and numbers will default to zero.

- *length*

Currently only valid for fixedstring fields, *length* specifies the length of the field in bytes. There is no default length; *length* must be specified for fixedstring fields.

- *unique*

If *unique* is specified with a true value, the field is defined as indexed, and an index has been created and is in existence for this field for the current table, a unique check will be performed on this field upon insertion into the speed table.

**Bug: Unique checks are not currently being performed as of 12/31/06.**

**Bug: String search matching functions don't yet work for fixedstrings and fixedstrings have not had a lot of use as of 12/31/06.**



# An Example Speed Table Definition

# 4

Example Speed Table Definition

This chapter provides an example speed table definition, explains it, and shows some basic usage of a speed table.

## Example Speed Table Definition

```
package require speedtable

CExtension animinfo 1.1 {

  SpeedTable animation_characters {
    varstring name indexed 1 unique 0
    varstring home
    varstring show indexed 1 unique 0
    varstring dad
    boolean alive default 1
    varstring gender default male
    int age
    int coolness
  }
}
```

- Speed tables are defined inside the code block of the *CExtension*.
- Executing this will generate table-specific C functions a Tcl C language extension named *Animinfo*, compile it and link it into a shared library.
- Multiple speed tables can be defined in one *CExtension* definition.
- No matter how you capitalize it, the package name will be the first character of your C extension name capitalized and the rest mapped to lowercase.

The name of the C extension follows the CExtension keyword, followed by a version number, and then a code body containing table definitions.

## Loading Your Speed Table-Generated C Extension

After sourcing in the above definition, you can do a `package require Animinfo` or `package require Animinfo 1.1` and Tcl will load the extension and make it available.

For efficiency's sake, we detect whether or not the C extension has been altered since the last time it was generated as a shared library, and avoid the compilation and linking phase when it isn't necessary.

Sourcing the above code body and doing a `package require Animinfo` will create one new command, *animation\_characters*, corresponding to the defined table. We call this command a *meta table* or a *creator table*.

`animation_characters create t` creates a new object, **t**, that is a Tcl command that will manage and manipulate zero or more rows of the *animation\_characters* table.

## One meta table can create many speed tables

You can create additional instances of the table using the meta table's *create* method. All tables created from the same meta table operate independently of each other, although they share the meta table data structure that speed table implementation code uses to understand and operate on the tables.

You can also say...

```
set obj [animation_characters create #auto]
```

...to create a new instance of the table (containing, at first, zero rows), without having to generate a unique name for it.

## Speed Table Basic Usage Examples

```
t set shake name "Master Shake" \  
  show "Aqua Teen Hunger Force"
```

This creates a new row in the speed table named **t**. Currently all rows in a speed table must have unique key value, which resides outside of the table definition itself. The key for this row is "shake".<sup>3</sup> The name and show fields in the row are set to the passed-in values.

---

<sup>3</sup> It feels a bit clumsy to have an external key like this, and we can pretty easily make the field be a part of the row itself, which seems better. It has generally proven useful to have some kind of unique key for each row although we can and do synthesize our own and if we're willing to write it, explicitly support tables with no unique keys at all.

We can set other fields in the same row:

```
t set shake age 4 coolness -5
```

And increment them in one operation:

```
% t incr shake age 1 coolness -1  
5 -6
```

I can fetch a single value pretty naturally...

```
if {[t get $key age] > 18} {...}
```

Or I can get all the fields in definition order:

```
puts [t get shake]  
{ } { } { } { } { } 1 male 5 -6
```

Forgot what fields are available?

```
% t fields  
id name home show dad alive gender age coolness
```

You can get a list of fields in array get format:

```
array set data [t array_get shake]  
puts "$data(name) $data(coolness)"
```

In the above example, if a field's value is null then the field name and value will not be returned by *array\_get*. So if a field can be null, you'll want to check for its existence using *array\_get\_with\_nulls*, which will always provide all the fields' values, substituting a settable null value (typically the empty string) when the value is null.

Want to see if something exists?

```
t exists frylock  
0
```

Let's load up our table from a file tab-separated data:

```
set fp [open animation_characters.tsv]  
t read_tabsep $fp  
close $fp
```

## Search

Search is one of the most useful capabilities of speed tables. Let's use search to write all of the rows in the table to a save file:

```
set fp [open save.tsv]
```

```
t search -write_tabsep $fp
close $fp
```

Want to restrict the results to a certain set of fields? Use the `-fields` option followed by a list of the names of the fields you want.

```
t search -write_tabsep $fp \
    -fields {name show coolness}
```

Sometimes you might want to include the names of the fields as the first line...

```
t search -write_tabsep $fp \
    -fields {name show coolness} \
    -with_field_names 1
```

Let's find everyone who's on the Venture Brothers show who's over 20 years old, and execute code for each result:

```
t search -compare {{= show "Venture Brothers" {> age 20}}} \
    -array_get data -code {
        puts $data
    }
```

## Additional meta table methods

- *animation\_characters info* - which currently does nothing (boring)
- *animation\_characters null\_value \N* - which sets the default null value for all tables of this table type to, in this case, `\N` (Hey, didn't you just get done saying the tables are independent of each other? Well... yeah, I did. We're going to make it settable on a per-table basis, too, and that may have been done already -- it's pretty easy to add.)
- *animation\_characters method foo bar* - this will register a new method named *foo* and then invoke the proc *bar* with the arguments being the name of the object followed by whatever arguments were passed.

For example, if after executing `animation_characters method foo bar` and creating an instance of the *animation\_characters* table named *t*, if you executed

```
t foo a b c d
```

...then proc *bar* would be called with the arguments "x a b c d".

**BUG - Tcl appears to examine a shared library name and stop at the first numeric digit in an apparently somewhat inadequate attempt to make sure it doesn't include shared library version numbers in the expected `*_Init` and `*_Safelnit` function names for the library being generated. Consequently when you're defining a C extension via the `CExtension` command, do not include any digits in your C extension's name.**

The generated C source code, some copied .c and .h files, the compiled .o object file, and shared library are written in a directory called `build` underneath the directory that's current at the time the CExtension is sourced, unless a build path is specified. For example, after the "package require speed table" and outside of and prior to the CExtension definition, if you invoke

```
Speed TableBuildPath /tmp
```

...then those files will be generated in the `/tmp` directory. (It's a bad idea to use `/tmp` on a multiuser machine, of course, but could be OK for a dedicated appliance or something like that.)

Note that the specified build path is appended to the Tcl library search path variable, `auto_path`, if it isn't already in there.

# Methods for Manipulating Speed Tables

# 5

Methods for Manipulating Speed Tables

This chapter enumerates all of the defined methods that are available to interact with Speed Tables, with examples.

Now the nitty gritty... The following built-in methods are available as arguments to each instance of a speed table:

*get, set, array\_get, array\_get\_with\_nulls, exists, delete, count, foreach, type, import, import\_postgres\_result, export, fields, fieldtype, needs\_quoting, names, reset, destroy, statistics, write\_tabsep, read\_tabsep, key, makekey, store, share, getprop, attach*

For the examples, assume we have done a "cable\_info create x"

- *set*

```
x set key ?-nocomplain? field value ?field value...?
```

or

```
x set key ?-nocomplain? keyValueList
```

The key is required and it must be unique. It can contain anything you want. It's not also an element of the table.

We may change this in the future to make it possible to have tables that do not require any keys (there is already a provision for this, though incomplete) and also to allow more than one key. But for now, lame or not, this is how it works, and as Peter says, for more than one key, you can always create some kind of compound key.

```
% x set peter ip 127.0.0.1 name "Peter da Silva" i 501
```

In the above example, we create a row in the **cable\_info** table named "x" with an index of "peter", an ip value of 127.0.0.1, a name of "Peter da Silva", and an "i" value of 501. All fields in the row that have not been set will be marked as null. (Also any field set with the null value will also be marked as null.)

```
% set values [list ip 127.0.0.1 name "Peter da Silva" i 501]
```

```
% x set peter $values
```

In this example, we specify the value as a list of key-value pairs. This is a natural way to pull an array into a speed table row:

```
% x set key [array get dataArray]
```

By default it is an error to attempt to set a field in a row that does not exist. However, if `-nocomplain` is specified as the first argument, such errors are suppressed, hence all matching fields are set and any keys that do not exist in the table are silently ignored. This is useful when an array contains some fields that you want to store in a speedtable row but may contain additional fields that you do not want to store but which, without `-nocomplain`, you'd have to remove from the array prior to invoking `set`.

- *store*

```
x store ?-nocomplain? field value ?field value...?
```

or

```
x store ?-nocomplain? keyValueList
```

Store is similar to “set”, but extracts the key from the provided fields. If the table does not have a field explicitly designated as a key, then the pseudo-field “\_key” is used. If the key is not present in the list, then the next auto-generated value (see `read_tabsep`) will be used.

Store returns the key used to store the list.

- *makekey*

```
x makekey field value ?field value...?
```

or

```
x makekey keyValueList
```

This simply calculates what the appropriate key value for the list would be.

For example, for a table where the field “ip” was a key:

```
x makekey {ip 10.2.3.1 name host1}
```

would return “10.2.3.1”

- *key*

Returns the name of the key field specified for the table, or “\_key” if none were specified.

- *fields*

“fields” returns a list of defined fields, in the order they were defined.

```
% x fields
```

```
ip mac name address addressNumber geos i j ij
```

- *field*

"field" returns information about the values that defined the field. You can use this command to retrieve all the key-value pairs that define a field.

Since we accept (and ignore) arguments to field definitions for keys we don't recognize, you can define your own key-value pairs in field definitions inside of speed table definitions and access them using this method.

Following the name of the field should be one of the keywords `getprop`, `properties`, or `proplist`. `Properties` will return the names of all of the properties as a Tcl list. `Proplist` will return the names and values of all the properties as a Tcl list, in what we would call "array set" format. `Getprop` will return the value associated with the key passed as an argument.

```
% $ctable field $fieldName proplist
```

```
default 1 name alive type boolean
```

```
% $ctable field $fieldName properties
```

```
default name type
```

```
% $ctable field $fieldName getprop default
```

- *get*

Get fields. Get specified fields, or all fields if none are specified, returning them as a Tcl list.

```
% x get peter
```

```
127.0.0.1 {} {Peter da Silva} {} {} {} 501 {} {}
```

```
% x get peter ip name
```

```
127.0.0.1 {Peter da Silva}
```

- *array\_get*

Get specified fields, or all fields if none are specified, in "array get" (key-value pair) format. Note that if a field is null, it will not be fetched.

```
% x array_get peter
```

```
ip 127.0.0.1 name {Peter da Silva} i 501
```

```
% x array_get peter ip name mac
```

```
ip 127.0.0.1 name {Peter da Silva}
```

- *array\_get\_with\_nulls*



Get specified fields, or all fields if none are specified, in "array get" (key-value pair) format. If a field contains the null value, it is fetched anyway. (Yes this should probably be an option switch to array\_get instead of its own method.)

```
% x array_get_with_nulls peter  
  
ip 127.0.0.1 mac {} name {Peter da Silva} address {}  
addressNumber {} geos {} i 501 j {} ij {}  
  
% x array_get_with_nulls peter ip name mac  
  
ip 127.0.0.1 name {Peter da Silva} mac {}
```

Note that if the null value has been set, that value will be returned other than the default null value of an empty Tcl object.

```
% cable_info null_value \N  
  
% x array_get_with_nulls peter  
  
ip 127.0.0.1 mac \N name {Peter da Silva} address \N  
addressNumber \N geos \N i 501 j \N ij \N  
  
% x array_get_with_nulls peter ip name mac  
  
ip 127.0.0.1 name {Peter da Silva} mac \N
```

- *exists*

Return 1 if the specified key exists, 0 otherwise.

```
% x exists peter  
  
1  
  
% x exists karl  
  
0
```

- *delete*

Delete the specified row from the table. Returns 1 if the row existed, 0 if it did not.

```
% x delete karl  
  
0  
  
% x set karl  
  
% x delete karl  
  
1  
  
% x delete karl  
  
0
```

- *count*

Return a count the number of rows in the table.

```
% x count
```

```
1
```

- *batch*

Take a list of speed table commands (minus the table name, as that's implicit), and invoke each element of the list as a method invocation on the current speed table.

A result list is constructed.

As each command within the batch is invoked, if the invocation is successful and no value is returned, nothing is added to the result list.

If the invocation is successful and a value is returned, a list is added to the result list containing two elements: the number of the element of the batch list and a sublist containing the Tcl result code (0) and whatever the result was that was returned.

If the invocation failed, a list is added to the result list, containing the element index, as above, but with the Tcl result code set to TCL\_ERROR (1) and the result portion is the error message returned.

```
% x batch {{set dean age 17} {incr dean age 1} {incr  
brock age foo}}  
  
{{1 {0 18}} {2 {1 {expected integer but got "foo"  
while converting age increment amount while processing  
key-value list}}}}
```

In this example, setting Dean's age to 17 produced no result. Incrementing it returned the incremented value (18), and trying to set Brock's age to a non-integer value recorded an error.

Note that errors in batched commands do not cause batch to return an error. It is up to the caller to examine the result of the batch command to see what happened.

"batch" will return an error in the event of bad arguments passed to it, the batch list being unparseable as a list, etc.

- *search*

Search for matching rows and take actions on them, with optional sorting. Search exploits indexes on fields when available, or performs a brute force search if there are no indexed fields available in the compare list. These indexes are implemented using skip lists.

## Brute-Force Search Is Brutally Fast

Search can perform brute-force multivariable searches on a speed table and take actions on matching records, without any scripting code running on an every-row basis.

On a modern 2006 Intel and AMD machines, speed table search can perform, for example, unanchored string match searches at a rate of sixteen million rows per CPU second (around 60 nanoseconds per row).

On the other hand, skip lists point to a future where there isn't any key that's external to the row -- that is, what would have been the external key would exist as a normal field in the row.

Whether you should use indexes (skiplists) or not depends on the characteristics of the table. On one of our test systems, inserting a row into the table takes about 2.3 microseconds, but a single index increases this to about 7 microseconds. On the other hand, an indexed search on that field may be  $O(\log N)$  on the number of rows in the table.

Search is a powerful element of the speed tables tool that can be leveraged to do a number of the things traditionally done with database systems that incur much more overhead.

```
$speedtable search ?-sort {?-?field..}? ?-fields  
fieldList? ?-glob pattern? ?-regexp pattern? ?-compare  
list? ?-countOnly 0|1? ?-offset offset? ?-limit limit?  
?-code codeBody? ?-write_tabsep channel? ?-key keyVar?  
?-get varName? ?-array_get varName? ?-ar-  
ray_get_with_nulls varName? ?-with_field_names 0|1?
```

- **-sort sortArg**

Sort results based on the specified field or fields. If multiple fields are specified, their precedence is in descending order. In other words, the first field is the primary search key.

If you want to sort a field in descending order, put a dash in front of the field name.

**Bug: Speed tables are currently hard-coded to sort null values “high”. As this is not always what one wants, an ability to specify whether nulls are to sort high or low will likely be added in the future.**

- **-fields fieldList**

Restrict search results to the specified fields.

If you have a lot of fields in your table and only need a few, using -fields to restrict retrieval to the specified fields will provide a nice performance boost.

Fields that are used for sorting and/or for comparison expressions do not need to be included in -fields in order to be examined.

- `-glob pattern`  
Perform a glob-style comparison on the key, excluding the examination of rows not matching.
- `-regexp pattern`  
Not currently implemented.
- `-countOnly 0|1`  
If 1, counts matching rows but does not take any action based on the count.
- `-offset offset`  
If specified, begins actions on search results at the "offset" row found. For example, if offset is 100, the first 100 matching records are bypassed before the search action begins to be taken on matching rows.
- `-limit limit`  
If specified, limits the number of rows matched to "limit".  
  
Even if used with `-countOnly`, `-limit` still works, so if, for example, you want to know if there are at least 10 matching records in the table but you don't care what they contain or if there are more than that many, you can search with `-countOnly 1 -limit 10` and it will return 10 if there are ten or more matching rows.
- `-write_tabsep channel`  
Matching rows are written tab-separated to the file or socket (or postgresql database handle) "channel".
- `-tab string`  
Specify the separator string for `write_tabsep` (default "\t").
- `-with_field_names 1`  
If you are doing `-write_tabsep`, `-with_field_names 1` will cause the first line emitted to be a tab-separated list of field names.
- `-key keyVar`
- `-get listVar`
- `-array arrayName`
- `-array_with_nulls arrayName`
- `-array_get listVar`
- `-array_get_with_nulls listVar`

- `-code codeBody`

Run scripting code on matching rows.

If `-key` is specified, the key value of each matching row is written into the variable specified as the argument that follows it.

If `-get` is specified, the fields of the matching row are written into the variable specified as the argument to `-get`. If `-fields` is specified, you get those fields in the same order. If `-fields` is not specified, you get all the fields in the order they were defined. If you have any question about the order of the fields, just ask the speed table with `$table fields`.

`-array_get` works like `-get` except that the field names and field values are written into the specified variable as a list, in a manner that *array get* can load into an array. I call this "array set" format. Fields that are null are not retrieved with `-array_get`.

`-array_get_with_nulls` pulls all the fields, substituting the null value (by default, an empty string) for any fields that are null.

Note it is a common bug to use `-array_get` in a `-code` loop, array set the returned list of key-value pairs into an array, and not *unset* the array before resuming the loop, resulting in null variables not being unset -- that is, from a previous row match, field x had a value, and in the current row, it doesn't.

If you haven't unset your array, and you "array get" the new result into the array, the previous value of x will still be there. So either unset (`-nocomplain` is a useful, not widely known optional argument to unset) or use `array_get_with_nulls`.

Better yet would be to just use `-array` or `-array_with_nulls`, both of which directly put the stuff in an array on your behalf and do the right thing with respect to null values.

`-array` sets field names and field values into the named array. Any fields that are null are specifically removed (unset) from the array.

Thus, if you use `-array` to and you with to access a field can be null, you need to check to see if the field exists (using `[info exists array(fieldName)]`, etc) before trying to look at its value.

If you don't want to do that, consider using `-array_with_nulls` instead.

`-array_with_nulls` sets field names and field values into the named array. Any fields that are null are set into the array as the null value (by default, an empty string), as set by the *null\_value* method of the creator table.

- `-compare list`

Perform a comparison to select rows.

Compare expressions are specified as a list of lists. Each list consists of an operator and one or more arguments.

When the search is being performed, for each row all of the expressions are evaluated left to right and form a logical "and". That is, if any of the expressions fail, the row is skipped.

Here's an example:

```
$speed table search -compare {{> coolness 50} \
    {> hipness 50}} ...
```

In this case you're selecting every row where coolness is greater than 50 and hipness is greater than 50.

Here are the available expressions:

- `{false field}`

Expression compares true if field's value is false. (For booleans, false. For shorts, ints and wides, false is 0 and anything else is true.

- `{true field}`

Expression compares true if field is true.

- `{null field}`

Expression compares true if field is null.

- `{nonnull field}`

Expression compares true if field is not null.

- `{< field value}`

Expression compares true if field less than value. This works with both strings and numbers, and yes, compares the numbers as numbers and not strings.

- `{<= field value}`

Expression compares true if field is less than or equal to value.

- `{= field value}`

Expression compares true if field is equal to value.

- `{!= field value}`

Expression compares true if field is not equal to value.

- `{>= field value}`  
Expression compares true if field is greater than or equal to value.
- `{> field value}`  
Expression compares true if field is greater than value.
- `{match field expression}`  
Expression compares true if field matches glob expression. Case is insensitive.
- `{match_case field expression}`  
Expression compares true if field matches glob expression, case-sensitive.
- `{notmatch field expression}`  
Expression compares true if field does not match glob expression. Case is insensitive.
- `{notmatch_case field expression}`  
Expression compares true if field does not match glob expression, case-sensitive.
- `{range field low hi}`  
Expression compares true if field is within the range of low <= field < hi.
- `{in field valueList}`  
Expression compares true if the field's value appears in the value list.  
  
The "in" search expression, when used as the first search term on an indexed field has very high performance, in particular with client-server ctables, as it is much faster to go find many rows in one query than to repeatedly cause a TCP/IP command/response roundtrip on a per-row basis.

**Bug:** As of 1/16/07 `in` only works with `search+`, as the first search term, when an index is defined on the field and is currently in existence. We expect to fully support `in` as any search term and on fields that don't have indexes in the future.

Examples:

Write everything in the table tab-separated to channel \$channel

```
$speed table search -write_tabsep $channel
```

Write everything in the table with coolness > 50 and hipness > 50:

```

$speedtable search -write_tabsep $channel \
    -compare {{> coolness 50} {> hipness 50}}
Run some code every everything in the table matching above:
$speedtable search \
    -compare {{> coolness 50} {> hipness 50}} \
    -key key -array_get data -code {
        puts "key -> $key, data -> $data"
    }

```

- `-poll_interval interval`

Perform an update operation every `interval` rows, to allow background processing to work in the background while a long search is going on.

- `-poll_code {code}`

Perform the specified code every `-poll_interval` rows. Errors from the code will be handled by the `berror` mechanism. If no poll interval is specified then a default (1024) is used.

- `search+`

Search+ is a (deprecated) synonym for search.

- `foreach`

DEPRECATED (use "search" instead)

```
x foreach varName ?pattern? codeBody
```

Iterate over all of the rows in the table, or just the rows in the table matching a string match wildcard, executing tcl code on each of them.

Example:

```
% x foreach key {
    puts $key
}
```

This is equivalent to:

```
% x search -key key -code {
    puts $key
}
```

- `incr`



Increment the specified numeric values, returning a list of the new incremented values

```
% x incr $key a 4 b 5
```

...will increment \$key's a field by 4 and b field by 5, returning a list containing the new incremented values of a and b.

- *type*

Return the "type" of the object, i.e. the name of the object-creating command that created it.

```
% x type
```

```
cable_info
```

- *import\_postgres\_result*

```
x import_postgres_result handle ?-nokeys? ?-nocomplain?
```

Given a *Pgtcl* result handle, *import\_postgresql\_result* will iterate over all of the result rows and create corresponding rows in the table, matching the SQL column names to the field names.

If the "-nocomplain" option is specified unknown columns in the result will be ignored.

If the "-nokeys" option is specified the key is derived from the key column specified for the table, or autogenerated as described in *read\_tabsep*.

This is extremely fast as it does not do any intermediate Tcl evaluation on a per-row basis.

How you use it is, first, execute some kind of query:

```
set res [pg_exec $connection "select * from mytable"]
```

(You can also use *pg\_exec\_prepared* or even the asynchronous *Pgtcl* commands *pg\_sendquery* and *pg\_sendquery\_prepared* in association with

```
pg_getresult -- see the Pgtcl documentation for more info.)
```

Check for an error...

```
if {[pg_result $res -status] != "PGRES_RESULT_OK"} {...}
```

...and then do...

```
x import_postgres_result $res
```

Importing PostgreSQL Results Is Pretty Fast

On a 2 GHz AMD64 we are able to import about 200,000 10-element rows per CPU second, i.e. around 5 microseconds per row. Importing goes more slowly if one or more fields of the speed table has had an index created for it.

- *fieldtype*

Return the datatype of the named field.

```
foreach field [x fields] {  
    puts "$field type is [x fieldtype $field]"  
}  
  
ip type is inet  
mac type is mac  
name type is varstring  
address type is varstring  
addressNumber type is varstring  
geos type is varstring  
i type is int  
j type is int  
ij type is long
```

- *needs\_quoting*

Given a field name, return 1 if it might need quoting. For example, varstrings and strings may need quoting as they can contain any characters, while integers, floats, IP addresses, MAC addresses, etc, do not, as their contents are predictable and their input routines do not accept tabs.

- *names*

Return a list of all of the keys in the table. This is fine for small tables but can be inefficient for large tables as it generates a list containing each key, so a 650K table will generate a list containing 650K elements -- in such a case we recommend that you use *search* instead.

This should probably be deprecated.

- *reset*

Clear everything out of the table. This deletes all of the rows in the table, freeing all memory allocated for the rows, the rows' hashtable entries, etc.

```
% x count  
652343
```

```
% x reset
% x count
0
```

- *destroy*

Delete all the rows in the table, free all of the memory, and destroy the object.

```
% x destroy
% x asdf

invalid command name "x"
```

- *share*
- *getprop*

Access information about the underlying shared memory associated with a shared memory table (see the section on shared memory, below).

- *attach*

Create an attachment for a shared reader table in a shared master table. Returns a set of *create* parameters to use to complete the attachment.

- *statistics*

Report information about the hash table such as the number of entries, number of buckets, bucket utilization, etc. It's fairly useless, but can give you a sense that the hash table code is pretty good.

```
% x statistics

1000000 entries in table, 1048576 buckets
number of buckets with 0 entries: 407387
number of buckets with 1 entries: 381489
number of buckets with 2 entries: 182642
number of buckets with 3 entries: 59092
number of buckets with 4 entries: 14490
number of buckets with 5 entries: 2944
number of buckets with 6 entries: 462
number of buckets with 7 entries: 63
number of buckets with 8 entries: 6
number of buckets with 9 entries: 0
```

number of buckets with 10 or more entries: 1

average search distance for entry: 1.5

- [write\\_tabsep](#)

DEPRECATED (use search -write\_tabsep)

```
x write_tabsep channel ?-glob pattern? ?-nokeys?  
?-with_field_names? ?-tab string? ?field...?
```

Write the table tab-separated to a channel, with the names of desired fields specified, else all fields if none are specified.

```
set fp [open /tmp/output.tsv w]  
x write_tabsep $fp  
close $fp
```

If the glob pattern is specified and the key of a row does not match the glob pattern, the row is not written.

The first field written will be the key, unless -nokeys is specified and the key value is not written to the destination.

If -with\_field\_names is specified, then the names of the fields will be the first row output.

If -tab is specified then the string provided will be used as the tab.

**BUG - We do not currently quote any tabs that occur in the data, so if there are tab characters in any of the strings in a row, that row will not be read back in properly. In fact, we will generate an error when attempting to read such a row. In most cases it should be possible to select a tab separator that does not occur in any field to avoid this.**

- [read\\_tabsep](#)

```
x read_tabsep channel ?-glob pattern? ?-nokeys?  
?-with_field_names? ?-tab string? ?-skip pattern?  
?field...?
```

Read tab-separated entries from a channel, with a list of fields specified, or all fields if none are specified.

```
set fp [open /tmp/output.tsv r]  
x read_tabsep $fp  
close $fp
```

The first field is expected to be the key (unless -nokeys is specified) and is not included in the list of fields. So if you name five fields, for example, each row in the input file (or socket or whatever) should contain six elements.

It's an error if the number of fields read doesn't match the number expected.

If the glob pattern is defined, it's applied to the key (first field in the row) and if it doesn't match, the row is not inserted.

If `-tab string` is specified, then the string provided will be used as the tab separator. There is no explicit limit on the length of the string, so you can use something like `-tab {%JULIE@ANDREWS%}` with `read_tabsep` and `write_tabsep` (or `search -write_tabsep`) to reduce the possibility of a conflict.

If `-skip pattern` is specified, then lines matching that pattern are ignored. This is sometimes necessary for files containing comments.

If `-with_field_names` is specified, the first row read is expected to be a tab-separated list of field names, and Speedtables will read that line and use its contents to determine which fields each of the following lines of tab-separated values will be stored as. (This is the counterpart to the `-with_field_names` argument to speedtables's `search` method when invoked with the `-write_tabsep` option.)

If `-nokeys` is specified, the first field of each row is not used as the key -- rather, the key is taken from the provided fields (as if `makekey` was called for each row), and if there is no key it is automatically created as an ascending integer starting from 0. The last key generated will be returned as the value of `read_tabsep`.

If you subsequently do another `read_tabsep` with `-nokeys` specified, the auto key will continue from where it left off. If you invoke the table's reset method, the auto key will reset to zero.

If you later want to insert at the end of the table, you need to use `store` rather than `set`.

`read_tabsep` stops when it reaches end of file OR when it reads an empty line. Since you must have a key and at least one field, this is safe. However it might not be safe with `-nokeys`.

The nice thing about it is you can indicate end of input with an empty line and then do something else with the data that follows.

- [index](#)

Index is used to create skip list indexes on fields in a table, which can be used to greatly speed up certain types of searches.

```
x index create foo 24
```

...creates a skip list index on field "foo" and sets it to for an optimal size of  $2^{24}$  rows. The size value is optional. (How this works will be improved/ altered in a subsequent release.) It will index all existing rows in the table

and any future rows that are added. Also if a *set*, *read\_tabsep*, etc, causes a row's indexed value to change, its index will be updated.

If there is already an index present on that field, does nothing.

```
x index drop foo
```

...drops the skip list on field "foo." if there is no such index, does nothing.

```
x index dump foo
```

...dumps the skip list for field "foo". This can be useful to help understand how they work and possibly to look for problems.

```
x index count foo
```

...returns a count of the skip list for field "foo". This number should always match the row count of the table (x count). If it doesn't, there's a bug in index handling.

```
x index span foo
```

...returns a list containing the lexically lowest entry and the lexically highest entry in the index. If there are no rows in the table, an empty list is returned.

```
x index indexable
```

...returns a (potentially empty) list of all of the field names that can have indexes created for them. Fields must be explicitly defined as indexable when the field is created with *indexed 1* arguments. (This keeps us from incurring a lot of overhead creating various things to be ready to index any field for fields that just couldn't ever reasonably be used as an index anyway.

```
x index indexed
```

...returns a (potentially empty) list of all of the field names in table x that current have an index in existence for them, meaning that index create has been invoked on that field.

# Special Notes On How To Make Searches Go Fast

# 6

Special Notes On How To Make Searches Go Fast

This chapter explains how to make Speed Table searches go as fast as possible.

An example of brute force searching that there isn't much getting around without adding fancy full-text search is unanchored text search. Even in this case, with our fast string search algorithm and quick traversal during brute-force search, we're seeing 60 nanoseconds per row or searching about sixteen million rows per CPU second on circa-2006 AMD64 machines.

Although many optimizations are being performed by the speed table compiler, further performance improvements can be made without introducing huge new complexities, perturbations, etc.

If you need to search for ranges of things, partial matches, straight equality of a field other than the key field, etc, you can use indexes and the "range", "=", and "in" compare functions to obtain huge search performance improvements over brute force, subject to a number of limitations: First, the table must have had an index created on that field using `$speedtable index create $fieldName`.

## Speed Table Query Optimizer

The Speed Table Query Optimizer has been rolled in to search, and `search+` has been deprecated. The "best" field in the query is used as the index, in this order:

"in" has the highest priority, but the field used MUST be the key field or an indexed field.

"=" has the next highest priority.

"<", "<=", or ">=" come next.

">" comes after these

All other searches are last priority.

In an ordered search, with an increasing sort, the sort field gets chosen when possible to avoid manually sorting the results after finding them.

# Client-Server Speed Tables

Client-Server Speed Tables

This chapter describes the rationale for creating a client-server interface to Speed Tables, explains the pluses and minuses of the current implementation, explains how to use the client interface, and gives an example of how to use it.

Tables created with Speed Tables, as currently implemented, are local to the Tcl interpreter that created them.

A version that uses shared memory and supports multiple readers is now available. It maintains the entire table, keys, and indexes in shared memory, and may be used when there is sufficient physical memory available. It operates locklessly and so does not support multiple writers.

Only the “search” command operates over the shared memory interface, all other commands use the client-server API.

Even with these limitations, client-server shared speed tables can be quite useful.

Early in our work it became clear that we needed a client-server way to talk to Speed Tables that was highly compatible with accessing Speed Tables natively.

The simplicity and uniformity of the speed tables interface and the rigorous use of key-value pairs as arguments to search made it possible to implement a Speed Tables client and server in around 500 lines of Tcl code. This code implements the Speed Table Transfer Protocol (STTP).

This implementation provides near-identical behavior for client-server Speed Tables as direct Speed Tables for *get*, *set*, *array\_get*, *array\_get\_with\_nulls*, *exists*, *delete*, *count*, *type*, *fields*, *fieldtype*, *needs\_quoting*, *names*, *reset*, *destroy*, *statistics*, and *search*.

The main exception is that it is not possible to call speedtable commands from within a code body running in a search, unless you use the shared-memory search to speed things up.



## Authentication, lack thereof

The current implementation of the speed table server does **no authentication**, so it is only appropriate for use behind a firewall or with a protection mechanism “in front of” it.

For instance, you might use your system’s firewall rules to prevent access to the ports speed table server is using (or you’re having it use) other than between the machines you designate. Alternatively you could add the TLS extension, do authentication and substitute SSL sockets for the plain ones -- Speed Tables wouldn’t even notice a difference.

There is a Tcl interpreter on the server side, pointing to the possibility of deploying server-side code to interact with Speed Tables<sup>4</sup>, although there isn’t any formal mechanism for creating and loading server-side code at this time.

Speed Tables’ *register* method appears to be a natural fit for implementing an interface to row-oriented server-side code invoked from a client.

Speed Tables can be operated in safe interpreters if desired, as one part of a solution for running server-side code, should you choose to take it on.

Once you start considering using Speed Tables as a way to cache tens of millions of rows of data across many tables, if the application is large enough, you may start to consider having machines basically serve as dedicated Speed Table servers.

Take generic machines and stuff them with the max amount of RAM at your appropriate density/price threshold. Boot up your favorite Linux or BSD off of a small hard drive, thumb drive, or from the network. Start up your Speed Tables server processes, load them up with data, and start serving speed tables at far higher performance than traditional SQL databases.

### Speed Table URLs

`sttp://foo.com/bar`

`sttp://foo.com:2345/bar`

`sttp://foo.com/bar/snap`

`sttp://foo.com:1234/bar/snap`

`sttp://foo.com/bar?moreExtraStuff=sure`

The default speed table client/server port is 11111. It can be overridden as above. There’s a host name, an optional port, an optional directory, a ta-

---

<sup>4</sup> Fairly analogous to stored procedures in a SQL database, Tcl code running on the server’s interpreter could perform multiple speed table actions in one invocation, reducing client/server communications overhead and any delays associated with it.

ble name, and optional extra stuff. Currently the optional directory and optional extra stuff are parsed, but ignored.

A typical server-side use of a speed table URL wildcards the hostname:

```
sttp://*:2345/bar
```

## Example Client Code

```
package require speedtable_client

remote_ctable sttp://127.0.0.1/dumbData t

t search -sort -coolness -limit 5 -key key -array_get_with_nulls data -code {
    puts "$key -> $data"
}
```

## Example Server Code

```
package require ctable_server

::ctable_server::register sttp://*/dumbData t
```

That's all there is to it. You have to allow the Tcl event loop to run, either by doing a *vwait* or by periodically calling *update* if your application is not event-loop driven, but as long as you do so, your app will be able to server out speedtables.

## Client-Server Performance

Performance of client-server speed tables is necessarily slower than that of native, local speed tables. Network round-trips and the Tcl interpreter being involved on both the client and server side for every method invoked on a remote speed table inevitably impacts performance.

That being said, a couple of techniques we will now explain can have a dramatic impact on client/server speed table performance.

## Batching Client-Server Speed Table Operations for Speed

Consider a case where you know you're going to set values in dozens to hundreds of rows in a table. You can batch up the sets into a single batch set command.

```
$remoteCtable set key1 var value ?var value...?
$remoteCtable set key2 var value ?var value...?
$remoteCtable set key3 var value ?var value...?
```

```

$remoteCtable batch {
    set key1 var value ?var value...?
    set key2 var value ?var value...?
    set key3 var value ?var value...?
}

```

In the second example, all of the set commands are sent over in a single remote speed table command, processed as a single batch by the speed table server (with no Tcl interpreter involvement in processing on a per-command basis inside the batch). A list is returned comprising the results of all of the commands executed. (See the batch method for more details.)

Most speed table commands can be batched, except for the search methods, the results of attempting such a thing being undefined. In particular, get, delete, and exists can be pretty useful.

### Using Client-Server Search To Get Many Rows At Once

Another common use of speed tables is to retrieve values from rows in some kind of loop. Perhaps something like...

```

foreach key $listOfRows {
    set data [$ctable get $key]
    ....
}

```

Unfortunately there is only a single channel for communication, and the server is single-threaded, so this doesn't work. Even if it did, every "get" would cause a network roundtrip to the speed table server handling that table. If we substitute a search for the above, we can get all the data for all the rows in a single roundtrip. The "in" compare method can be particularly useful for this...

```

$ctable search -compare {in key $listOfRows} -array_with_nulls
data {
    ....
}

```

### STTP does not support NULLs.

Note that -array\_with\_nulls retrieves null fields. STTP passes rows around internally as token separated files, and hence when used in client-server speed tables there is no equivalent to -array or -array\_get.

Because tab-separated data doesn't have an out-of-band facility for communicating that a field is null, null values must be communicated in-band.

## STAPI support for ctable\_server

The success of the ctable\_server led to the creation of a generic URI-based API for the ctable server and for other ctable-compatible objects and classes. This API, STAPI, allows ctables, the ctable server, and other compatible objects to be used interchangeably by applications.

To open a table using the STAPI you need to package require any packages needed for the STAPI connection method you need, then call

```
::stapi::connect method://server_spec/table_spec
```

For the speedtable server, the method is ctable: (as above) or sttp: (speed tables transfer protocol), and the URI syntax is exactly the same as in remote\_speedtable

As a special case, when the URI is not in URI format it is assumed to be the name of an already opened ctable.

STAPI connection methods already defined also include sql, which provides direct access to PostgreSQL tables through pgsq as if they were ctables.

STAPI is described in more detail in section 9.

# Shared Memory Speed Tables

# 8

## Shared Memory Speed Tables

Client-server speed tables can take a fairly big performance hit, as a sizable amount of Tcl code gets executed to make the remote speed table function like a local one.

While they're still pretty fast, server actions are inherently serialized because of the single-threaded access model afforded using standard Tcl fileevent actions within the Tcl event model.

When the speed table resides on the same machine as the client, and particularly in this era of relatively inexpensive multiprocessor systems, it would be highly desirable for a client to be able to access the speed table directly through shared memory, bypassing the server entirely.

This work was undertaken in the summer of '07 by Peter da Silva. The goal was to provide a way for same-server clients to access the speed table through shared memory while retaining the ability to build and use speed tables without using shared memory at all.

Tricky synchronization issues surfaced immediately. For instance, what should we do if a row gets changed or added while a search is being performed? We don't want to completely lock out access to the table during a search. Thus we have to really deal with database updates during searches, which raise referential integrity issues and garbage collection / dangling pointer issues. Many searches, such as ones involving results sorting, collecting a set of pointers to the rows that have matched. Those rows cannot disappear behind search's back.

Also the code was already in heavy production with tables containing tens of millions of rows. This work had to be rock solid or it wouldn't be usable.

To simplify the problem, we decided to funnel writes through the client/server mechanism and only allows reads and searches to occur through shared memory.

Our approach is to maintain metadata about in-progress searches in shared memory and have a cycle number that increases as the database is updated. When a search begins, the client copies the current cycle number to a word in shared memory allocated for it by the server. As normal activity causes rows to be modified, updated, or deleted by the server the cycle they were modified on is stored in the row. If rows (or any other shared memory object, such as strings)

are deleted, they are added to a garbage pool along with the current cycle, but not actually freed for reuse until the server garbage collects them on a later cycle.

If the client detects that a row it's examining has been modified since it started its search, it restarts the search operation. The server makes sure to update pointers within shared memory in an order such that the client will never step into a partially modified structure. This allows the whole operation to proceed without explicit locks, so long as pointer and cycle updates are atomic and ordered.

Garbage collection is performed by locating deleted memory elements that have a cycle number is lower than the cycle number of any client currently performing a search.

New options to the ctable "create" command:

- *\$ctable create name master list*

Creates a new master table based on the parameters in the list.

name tablename (optional)

The shared memory segment has a small write-once symbol table that is used to locate individual ctables and other objects in shared mem.

file filename (required)

Multiple tables can be mapped in the same file, distinguished by the ctable name or the name provided in the "name" option.

size bytes (required)

Used to create the file, or if it's already mapped it checks if it's at least this big.

flags list

The only shared memory flags implemented are sync/nosync (default nosync) and core/nocore (default core).

- *\$ctable create name reader list*

The list provided is collected from the master table (already opened in another process) through the attach command (below).

This attaches to an existing shared memory segment based on the information in the list, then searches for the reader cycle tagged by the process ID provided to attach, and creates a reader-mode ctable. This table contains a pointer to the master ctable in shared memory, data copied from the master, and other bookkeeping elements.

New ctable commands:

- *\$ctable attach pid*

Only valid for a master shared ctable, Creates a cycle entry for the process pid, and returns a list of parameters that describe how to attach to the ctable. Currently {file \$file name \$name}, where "file" is the file to map and "name" is the name of the shared ctable in the directory.

- *\$ctable getprop*

With no names, returns a name-value list of properties of the ctable, whatever is needed for reflection.

Currently type, extension, and key. These are needed for the STAPI glue for shared tables.

- *\$ctable getprop name ...*

With names, returns a list of only those properties.

- *share command [\$ctable share] ?options?*

- *\$ctable share command ?options?*

These are equivalent ways of doing the same thing. The "share" extension actually stands apart from the ctable extension and Tcl. It provides the shared memory segments and handles memory allocation from the segments. It's unlikely to be useful to use the explicit share form except in internal debugging. The rest of the doc assumes you're using "\$ctable share command".

The following commands are meaningful for ctable shares:

names

Returns a list of named objects in the share. These are not necessarily ctables, they may be string variables or objects created by other libraries.

set name string (*master only*)

Sets a shared string variable, for passing additional environment or context to readers.

get name

Gets the value of a string set with "set".

info

Returns some internal information about the share in a name-value list. The data includes size, flags, name, whether you're the creator (*master*), and filename.

#### pools

Returns a list of information for fixed size memory pools in the shared segment. There will be at least two pools, the garbage pool (containing elements that have been freed but are still in use by at least one reader) and one pool the size of a ctable row is set up for each ctable. For each pool it will return the size of elements that the pool will manage, how many elements in each chunk of elements allocated at once, the total number of chunks allocated, and the number of free elements: {element\_size elements\_per\_chunk chunks free\_elements}

#### pool element\_size elements\_per\_chunk max\_chunks

Creates a pool for objects of element\_size bytes, allocated in elements\_per\_chunk chunks, up to a maximum of max\_chunks. If max\_chunks is zero it will extend to pool to the limit of the shared segment if necessary.



## STAPI support for shared memory tables:

```
package require st_shared
::stapi::connect shared://port/table ?options?
```

Options:

- build path ... directory containing the generated ctable package.

Connect to a ctable on localhost as a ctable\_server client, and then open a parallel shared memory client for the same ctable. These connections are hidden behind a STAPI wrapper, so all ctable commands can be used: shared memory will be used for read-only “search” commands, and the ctable\_server TCP connection will be used for all other commands.

Server example:

```
top_brands_nokey_m create m master file sharefile.dat
[...]
::ctable_server::register ctable://*:1616/master m
[...]
if !$tcl_interactive { vwait die }
```

This is just like a normal ctable server, except that the ctable itself is a shared memory master table.

Client example:

```
package require st_shared
[...]
# Connect to the server, using the shared ctable
# extension created by the server in the directory
# “build”. Returns a stapi object.
set r [::stapi::connect shared://1616/master -build
build]

# This command is performed using shared memory.
$r search -compare {%= name phred}} -key k -code {puts
$k}

# This command is performed using TCP
$r set fred $row

# Close the reader and disconnect from the server.
$r destroy
```

# stapi - Speed Table API

stapi - Speed Table API

STAPI allows the speedtables API, originally implemented in ctables, to be used for a variety of table-like objects. This includes remote ctables through ctable\_server and SQL databases. There are two main sets of routines in STAPI, and they're not normally used together.

- `st_server`, a set of routines for automatically creating a ctable from an SQL table as a local read\_only cache for the table, or as a workspace to be used for preparing rows to be inserted into the table. It's normally used in a ctable\_server task providing a local cache for client processes.
- `st_client`, which provides the general interface for creating STAPI objects identified by URIs.

Normally,

## st\_server

```
package require st_server
::stapi::init ?options?
```

Options:

```
-dir work_directory
```

Root of directory tree for the ctables

```
-mode mode
```

Octal UNIX mode bits for new directories

```
-conn connection
```

Pgsql connection (if not specified, assumes DIO is being used and a DIO object named DIO exists and has already been connected to the database)

```
-cache minutes
```

How long to treat a cached tsv file as "good"

```
::stapi::init_ctable name table_list where_clause ?columns | column...?
```

Initialize a cached speed table based on one or more SQL tables. If necessary, this builds a ctable based on the columns, and generates new SQL to read the table.

Parameters:

`name`

base name of speed table

`table_list`

list of SQL tables to extract data from. If it's empty then use the base name of the speed table as the name of the SQL table.

`where_clause`

An optional SQL "WHERE" clause to limit the rows selected into the speed table, or an empty string

`columns`

list of column definitions.

At least two columns must be defined -- the first is the speed table key, the rest are the fields of the ctable. If there is only one "column" argument, it's assumed to be a list of column arguments.

Column entries are each a list of

{field type expr ?name value?...}

`field`

field name

`type`

SQL type

`expr`

SQL expression to derive the value

`name value`

speed table arguments for the field

(Only the field name is absolutely required.)

If the type is missing or blank, it's assumed to be *varchar*. If the expression is missing or blank, it's assumed to be the same as the field name.

In most cases the list of column definitions can be created by querying the SQL database itself using `from_table`:

```
::stapi::from_table table_name keys ?-option value?
```

Generate a column list for `init_ctable` by querying the SQL database for the table definition.

`keys`

a list of columns that define the key for the table

Keys can be empty, to allow you to combine `from_table` lists with an appropriate "WHERE" clause to use `init_ctable` to create a view that spans tables.

Options:

`-with column`

Include column name in table. If *any* `-with` clauses are provided, only the named columns will be included.

`-without column`

Exclude column name from table. You must not provide both `"-with"` and `"-without"` options.

`-index column`

Make this column indexable. The index will actually be created after the cache is loaded.

`-column {name type ?sql? ?args}`

Add an explicit derived column. This can be used for the creation of ctables from SQL tables that have multi-column keys.

`-table name`

If specified, generate implicit column-name as "table.column" in the SQL. This allows for the cache to be created from a query on more than one table.

`-prefix text`

If specified, prefix column names with "\$prefix"

```
::stapi::open_cached name ?pattern? ?-opt val?...
```

Open an initialized speed table, maintaining a local cache of the underlying SQL table in a .tsv file in the workdir.

Options

`-pat pattern`

Only read lines matching the pattern from the cache, if the cache is good. This is an optimization to avoid reading the entire table into memory when only a part of the table will be used. If the cache is old or missing, then the entire table will still be read into memory.

`-time cache_timeout`

Override the default cache timeout.

`-col name`

Name of column in the table that contains the `last_changed` time of each entry, if any. This is used as an optimization to only load modified lines when the schema supports that.

`-index field_name`

Name of a field to create an index on. Multiple `-index` entries are allowed.

`::stapi::refresh_ctable ctable ?last_read? ?err?`

Update new rows from SQL for speed table `ctable`.

If `last_read` is non-zero, use that rather than last modify time of the cache file.

If `err` is provided, it will return success or failure of the SQL request and put the error in `$err`, otherwise it will generate a Tcl error for SQL errors.

This uses the parameters set up in `open_cached`, and if there is no column in the table that can be used to determine the last change time, then the whole table will be re-read.

`::stapi::save_ctable ctable ?tsv_file?`

Save a table locally on disk. If the `tsv_file` is provided, it writes to that file. If not, it locates and locks the existing tsv file for the table, writes it, and unlocks it. This does not save the table back to the SQL data source.

`::stapi::remove_tsv_file table_name`

`::stapi::remove_tcl_file table_name`

Remove the cached **tcl** or **tsv** files, which will force the cache to be reread (if the **tsv** file is missing) or reconstructed using SQL queries (if the **tcl** file is missing). These are not normally used directly, but are available if the table is known to be out of date.

`::stapi::open_raw_ctable name`

Open an initialized speed table (as in `open_cached`) but don't fetch anything from SQL. This is used internally by `open_cached`, and is also useful for setting up temporary tables and workspaces.

## st\_client

st\_client implements that ::stapi::connect front end for ctables and other speedtable API objects.

```
::stapi::connect uri ?-name value...?
```

Connect to a speed table server or other database providing a speed table interface via a URI.

Returns an open speed table.

Options:

```
-key col
```

Define the column used to generate the key.

```
-keys {col col ...}
```

Define the columns used to generate the key.

```
-keysep "string"
```

Define the separator used to build the key.

One of -key or -keys/-keysep should be provided. Depending on the underlying object, -keys may not be compatible and STAPI will need to create a wrapper function.

If neither is provided, some STAPI capabilities may not be available.

```
::stapi::register method transport_handler
```

register a transport method for ::stapi::connect

At this time the following methods are defined in STAPI:

### Using a ctable server via sttp (client/server)

```
package require st_client
```

```
sttp://[host:port]/[dir/]table[/stuff][?stuff]
```

```
(or ctable://[host:port]/[dir/]table[/stuff][?stuff] )
```

### Using a ctable server via sttp (shared memory)

```
package require st_shared
```

```
shared://port/[dir/]table[/stuff][?stuff]
```

Access a speed table server on localhost, using shared memory for the "search" method and sttp: for other methods.

The speed table must reside on the same machine for shared memory table access to be used. Concurrent access and update of shared memory speed tables is supported and provides a mechanism to use multiple processors to access a table concurrently. Like, really concurrently, whereas pure client/server table access is inherently single threaded.

The ctable built by the server must be in auto\_path, or in the directory defined by the "-build" option.

### Using a PostgreSQL database directly

```
package require st_client_pgsql
```

```
::stapi::init_conn connection
```

```
::stapi::init_DIO ?db? ?user?
```

```
sql://connection/table[/col[:type]/col...][?param&param...]
```

Create a stapi interface to a PostgreSQL table

connection:

Not implemented yet, will be:

[user[:password]]@[host:]database

cols:

If no keys defined, first column is assumed to be the key.

params:

column=sql\_code

Define the SQL required to perform the selection



`_key=sql_code`

`_key=column`

Define the SQL for the key. If this is a simple column name no explicit “\_key” will be created.

`_keys=column:column:...`

Define the key in terms of a list of columns.

This uses the methods defined in `st_server`.

Examples:

**`sql:///users?_key=login`**

Pull in all the columns from “users”, using login as the key.

**`sql:///users/login/password`**

Pull in login and password from “users”, using login as the key.

## Using an already opened speed table

`package require st_client`

`name` (no method)

If the URI is not URI format, it assumes it's an object that provides stapi semantics already... typically a `ctable`, an already-opened `ctable_client` connection, or the result of a previous call to `::stapi::connect`. It queries the object using the `methods` command, and if necessary creates a wrapper around the `ctable` to implement the extra methods that STTP provides.

Required methods to avoid the creation of a wrapper:

`key`

`makekey`

## stapi::extend - the sttp API extensions to ctables

These extensions may be required for packages like `STdisplay`, but may not be provided by all speedtable-compatible packages, so `::stapi::extend` can create a wrapper object when needed.

`::stapi::extend::connect` object key-list ?-keysep string?

This is also called internally by `::stapi::connect` if the “-key” or “-keys” option is provided.

If the `_object_` was created by `::stapi::extend::connect`, or if it can use the `methods` call to determine that the object provides all the necessary methods, then the STAPI object is returned immediately. That makes it always safe to use this on an opened speedtable. but it can not be used to change the parameters of an existing STAPI object.

Otherwise, this behaves identically to calling `::stapi::connect` with the `-key/-keys` argument, and creates a wrapper object that understands at least the `key`, `makekey`, and `store` methods.

# STDisplay - Speed Table Display Functions for the World Wide Web

# 10

stDisplay - Display Functions for the WWW

From the speed table API, *stDisplay*, speed table display functions for the world wide web. This code is derived from Rivet's **diodisplay.tcl**.

```
set display [::STDisplay #auto ?-confvar value? ... \
    -table table ?-keyfields key_list?]
    or
set display [::STDisplay #auto ?-confvar value? ... \
    -uri uri]
```

One of `-table` or `-uri` must be provided.

`-table` - use already opened table.

If the table isn't STAPI-compatible it will use `::stapi::extend::connect` to wrap it and in this case `keyfields` must be provided.

`-uri` - Use `::stapi::connect` to connect to a ctable from a URI.

Any valid STAPI URI can be used.

Other config variables:

`-debug` 0-9

Set debug level

`-csvfile` filename

Name of CSV file to generate.

`-csvredirect` ?0|1?

If enabled, then the CSV file is downloaded immediately using a redirect.

`-key` field\_name

Field to use as a key

`-title` string

Title of table.

`-arrows {up down}`

Symbols or HTML fragments to use for ascending and descending sort of a table column.

`-pagesize integer`

Number of rows displayed per page.

`-functions {Search List Add Edit Delete Details}`

Functions shown on the top of the display.

`-rowfunctions {Details Edit Delete}`

Functions shown at the end of the row.

## Methods for `$display`

`$display show`

Render all HTML for the page. Destroys `$display`.

`$display destroy`

Destroys `$display`

`$display state`

Returns a list of name-value pairs that represent the current state of the query, as CGI variables, for populating external links. The counterpart to `hidden`

`$display functions {?Add? ?Show? ...}`

Equivalent to the `-functions` configuration variable - enable global functions, to control the functions available in the search bar. The possible functions are *Search*, *List*, *Add*, *Edit*, *Delete* and *Details*.

`$display rowfunctions {?Edit? ?Delete? ...}`

Equivalent to the `-rowfunctions` configuration variable - enable row functions, to control the functions available in the search bar. The possible functions are *Edit*, *Delete* and *Details*.

`$display field name ?-configvar value?...`

Define a field

`$display alias name ?field?`

Define an alias for an existing field. Aliases are used to create multiple filtered columns based on the same field.

`$display filter name ?proc? ?column...?`

Filter column *name* through [proc *\$value\_of\_field* ...] before displaying. If any columns are provided the values of these columns will be appended to the filter.

\$display **csvfilter** name ?proc? ?column...?

Filter column *name* through [proc *\$value\_of\_field*] on generating a CSV file, like **filter**.

\$display **unfilter** name ?proc?

Filter entered text through [\$proc text] before using to search table. This allows the user to enter (for example) a device name for a field that expects an IP address.

\$display **order** name \$order

Set sort order for field.

\$display **case** name \$case

Set whether the column can be matched uppercase or lowercase when doing a case-independent match.

\$display **hovertext** name \$text

Set text to be displayed when hovering the cursor over the title of the column.

\$display **blankval** name \$blank

Value to treat as a null value and display as an empty cell.

\$display **limit** \$limit

Set search terms to limit the displayed portion of the ctable. The limit is a list of {column\_name value ?column\_name value?...}.

\$display **attributes** name \$attribute\_list

Set attributes (eg "bgcolor=blue") for field

\$display **hidden** \$cgivar \$value

This is used to pass any additional CGI variables the page will require through the links generated by STDisplay. This method may be invoked for each value

# C Code Generated and C Routines Created

# 11

C Code Generated and C Routines Created

This chapter describes internal implementation details of Speed Tables. You can skip this section unless you're interested in finding out how Speed Tables work internally.

(There is a better interface than this for all but the lowest-level access code. You can interact with any speed table, regardless of its composition, by making standardized C calls via the speed table's methods and speed table's creator table structures. It's not documented yet but you can study **speedtable\_search.c**, where it is used extensively, and **speedtable.h**, where those structures are defined.)

The row format is not guaranteed to be the same between point releases of speed tables. However, fields you define will be accessible with the name you defined for them and of the data type corresponding to what you defined, regardless of the release, from the first version to the present and for the foreseeable future.

For the above cable\_info table defined, the following C struct is created:

```
struct cable_info {
    TAILQ_ENTRY(cable_info)  _link;
    struct in_addr            ip;
    struct ether_addr        mac;
    char                     *name;
    int                      _nameLength;
    char                     *address;
    int                      _addressLength;
    char                     *addressNumber;
    int                      _addressNumberLength;
    char                     *geos;
    int                      _geosLength;
    int                      i;
    int                      j;
```

```

long                ij;
struct Tcl_Obj      *extraStuff;
unsigned int        _ipIsNull:1;
unsigned int        _macIsNull:1;
unsigned int        _nameIsNull:1;
unsigned int        _addressIsNull:1;
unsigned int        _addressNumberIsNull:1;
unsigned int        _geosIsNull:1;
unsigned int        _iIsNull:1;
unsigned int        _jIsNull:1;
unsigned int        _ijIsNull:1;
unsigned int        _extraStuffIsNull:1;
};

```

Note that varstrings are *char \** pointers. We allocate the space for whatever string is stored and store the address of that allocated space. Fixed-length strings are generated inline.

The null field bits and booleans are all generated together and should be stored efficiently by the compiler. We rely on the C compiler to do the right thing with regards to word-aligning fields as needed for efficiency.

You can examine the C code generated -- it's quite readable. If you didn't know better, you might think it was written by a person rather than a program. (Several times when working on speed tables I've started editing the generated code rather than the code that's generating it, by mistake.)

Each table-defining command created has a *CTableCreatorTable* associated with it, for example:

```

struct CTableCreatorTable {
    Tcl_HashTable      *registeredProspeed tablePtr;
    long unsigned int  nextAutoCounter;

    int                nFields;
    int                nLinkedLists;

    CONST char         **fieldNames;
    Tcl_Obj             **nameObjList;
    int                *fieldList;
    enum ctable_types  *fieldTypes;
    int                *fieldsThatNeedQuoting;

    struct ctableFieldInfo **fields;

    void (*make_empty_row) ();
    int (*set) (Tcl_Interp *interp, struct CTableTable *ctable, Tcl_Obj *dataObj
, void *row, int field, int indexCtl);
    int (*set_null) (Tcl_Interp *interp, struct CTable *ctable, void *row,
int field, int indexCtl);

    Tcl_Obj *(*get) (Tcl_Interp *interp, void *row, int field);

```

```

    CONST char *(*get_string) (const void *pointer, int field, int *lengthPtr, T
    cl_Obj *utilityObj);

    Tcl_Obj *(*gen_list) (Tcl_Interp *interp, void *pointer);
    Tcl_Obj *(*gen_keyvalue_list) (Tcl_Interp *interp, void *pointer);
    Tcl_Obj *(*gen_nonnull_keyvalue_list) (Tcl_Interp *interp, void *pointer);
    int (*lappend_field) (Tcl_Interp *interp, Tcl_Obj *destListObj, void *p, int
    field);
    int (*lappend_field_and_name) (Tcl_Interp *interp, Tcl_Obj *destListObj, voi
    d *p, int field);
    int (*lappend_nonnull_field_and_name) (Tcl_Interp *interp, Tcl_Obj *destList
    Obj, void *p, int field);
    void (*dstring_append_get_tabsep) (char *key, void *pointer, int *fieldNums,
    int nFields, Tcl_DString *dsPtr, int noKey);

    int (*search_compare) (Tcl_Interp *interp, struct CTableSearchStruct *search
    Control, void *pointer, int tailoredWalk);
    int (*sort_compare) (void *clientData, const void *hashEntryPtr1, const void
    *hashEntryPtr2);
};

```

The registered proc table is how we handle registering methods, and the *nextAutoCounter* is how we can generate unique names for instances of the table when using "#auto".

*nFields* is the number of fields defined for the row, while *nLinkedLists* says how many doubly linked lists are included in each row. (The first doubly linked list is used by Speed Tables to link all rows of a table together; the rest are created for linking into index entries for each field that is defined as indexable.)

*fieldNames* is a pointer to an array of pointers to the name of each field, while *nameObjList* is a pointer to an array of pointers of Tcl objects containing the names of each field. By generating these once in the meta table, they can be used all over the place, by each speed table created by the meta table in many places, sharing these objects and neither incurring the memory or CPU overhead of constantly instantiating new Tcl objects from the name string whenever field names are needed.

*fieldList* is a pointer to an array of integers corresponding to the field numbers. Guess what? If there are six fields it will contain {0, 1, 2, 3, 4, 5}. The thing is we can feed it to routines we have that take such a list when the user has not told us what fields they want. *fieldTypes* are an array of data type numbers for each field. (Data type numbers are defined in speed table.h.) *fieldsThatNeedQuoting* is an array of ints, one for each field, saying if it needs quoting or not.

A number of the fields defined above are being consolidated into the *CTableFieldInfo struct*, which is defined for each field and contains the field name, name object, field number, type number, whether or not it needs quoting, its compare function (for indexing the the like, something we generate for each field), and its index number (which index of the array of doubly linked list elements built into each row), if indexed, else -1.



Finally, a number of pointers to functions to do things to the speed table are defined. This is cool stuff. As I began to code the complex sorting and indexing code, it started getting hard to keep my head wrapped around it all. Trying to custom-generate all that search code made complicated code even more complicated. Standardizing the search code to not be custom generated at all and to access the custom-generated aspects of the different Speed Tables through these function pointers.

Function pointers are provided to create an empty row, set a field of a row to a value, set a field of a row to null, get the native value of a field from a row as a Tcl object, and get a string representation of a field from a row. Additional function pointers are provided to get the contents of a row as a Tcl list and as a key-value Tcl list, with or without null values, to append the contents of a field to a list, to append the name of a field and the contents of a row's field to a list, and some other stuff like that.

Each instance of the table created with "create" has a speedtableTable associated with it:

```
struct CTableTable {
    struct CTableCreatorTable    *creatorTable;
    Tcl_HashTable                *keyTablePtr;
    Tcl_Command                  commandInfo;
    long                         count;
    jsw_skip_t                   **skipLists;
    struct ctable_baseRow        *ll_head;
    int                          nLinkedLists;
};
```

This contains a pointer to the meta table (creatorTable), a hash table that we use to store and fetch keys, a command info struct that we use to delete our created command from the Tcl interpreter when it's told to destroy itself, the row count, a pointer to an array of pointers to skip lists, one for each field that has an index defined for it (it's NULL otherwise).

A skip list for an indexed field can be walked to do a walk ordered by that field, as opposed to the pseudo-random ordering provided by walking the hash table or the last-thing-added-is-at-the-front ordering of "linked list zero", the linked list that all rows in a table are in.

Next, the number of fields is defined, the field names as an array of pointers to character strings and an enumerated type definition of the fields:

```
#define CABLE_INFO_NFIELDS 10
static CONST char *cable_info_fields[] = {
    "ip",
    "mac",
    "name",
```

```

    "address",
    "addressNumber",
    "geos",
    "i",
    "j",
    "ij",
    "extraStuff",
    (char *) NULL
};

```

```

enum cable_info_fields {
    FIELD_CABLE_INFO_IP,
    FIELD_CABLE_INFO_MAC,
    FIELD_CABLE_INFO_NAME,
    FIELD_CABLE_INFO_ADDRESS,
    FIELD_CABLE_INFO_ADDRESSNUMBER,
    FIELD_CABLE_INFO_GEOS,
    FIELD_CABLE_INFO_I,
    FIELD_CABLE_INFO_J,
    FIELD_CABLE_INFO_IJ,
    FIELD_CABLE_INFO_EXTRASTUFF
};

```

The types of each field are emitted as an array and whether or not fields need quoting:

```

enum ctable_types cable_info_types[] = {
    CTABLE_TYPE_INET,
    CTABLE_TYPE_MAC,
    CTABLE_TYPE_VARSTRING,
    CTABLE_TYPE_VARSTRING,
    CTABLE_TYPE_VARSTRING,
    CTABLE_TYPE_VARSTRING,
    CTABLE_TYPE_INT,
    CTABLE_TYPE_INT,
    CTABLE_TYPE_LONG,
    CTABLE_TYPE_TCLOBJ
};

```

```

int cable_info_needs_quoting[] = {
    0,
    0,
    1,
    1,
    1,

```

```

    1,
    0,
    0,
    0,
    1
};

```

A *setup routine* is defined that is automatically run once when the extension is loaded, for example, `cable_info_setup` creates some Tcl objects containing the names of all of the fields and stuff like that.

An *init routine*, for example, `cable_info_init`, is defined that will set a newly malloc'ed row to default values (Defaults can be specified for most fields. If a field does not have a default, that field's null bit is set to true.)

For efficiency's sake, we have a base copy that we initialize the first time the init routine is called and then for subsequent calls to initialize a row we merely do a structure copy to copy that base copy to the pointer to the row passed.

A delete routine is defined, for instance, `cable_info_delete`, that will take a pointer to the defined structure and free it. The thing here is that it has to delete any varstrings defined within the row prior to freeing the row itself.

\* **\_find** takes a pointer to the StructTable corresponding to the speed table, for instance, `cable_infoStructTable` and a char \* containing the key to be looked up, and returns a pointer to the struct (in the example, a struct `ctable_info *`) containing the matching row, or NULL if none is found.

\* **\_find\_or\_create** takes a pointer to the StructTable, a char \* containing the key to be looked up or created, and a pointer to an int. If the key is found, a pointer to its row is returned and the pointed-to int is set to zero. If it is not found, a new entry for that name is created, an instance of the structure is allocated and initialized, the pointed-to int is set to one, and the pointer to the new row is returned.

A \* **\_obj\_is\_null** routine is defined, for instance `cable_info_obj_is_null` that will return a 1 if the passed Tcl object contains a null value and zero otherwise.

\* **\_genlist** (`cable_info_genlist`), given a pointer to a Tcl interpreter and a pointer to a row of the corresponding structure type will generate a list of all of the fields in the table into the Tcl interpreter's result object.

\* **\_gen\_keyvalue\_list** does the same thing except includes the names of all the fields paired with the values.

\* **\_gen\_nonnull\_keyvalue\_list** does the same thing as `*_gen_keyvalue_list` except that any null values do not have their key-value pair emitted.

**\*\_set** (cable\_info\_set) can be used from your own C code to set values in a row. It takes a Tcl interpreter pointer, a pointer to a Tcl object containing the value you want to set, a pointer to the corresponding structure, and a field number from the enumerated list of fields.

It handles detecting and setting the null bit as well.

**\*\_set\_fieldobj** is like \*\_set except the field name is contained in a Tcl object and that field name is extracted and looked up from the field list to determine the field number used by \*\_set.

**\*\_set\_null** takes a row pointer and a field number and sets the null bit for that field to true. Note there is no way to set it to false except to set a value into a field as simply clearing the bit would be an error unless some value was written into the corresponding field.

**\*\_get** fetches a field from a table entry and returns a Tcl object containing that field. It takes a pointer to the Tcl interpreter, a pointer to a row of the structure, and a field number. If the null bit is set, the null value is returned.

Even though it is returning Tcl objects, it's pretty efficient as it passes back the same null object over and over for null values and uses the correct **Tcl\_New\*Obj** for the corresponding data type, hence ints are generated with **Tcl\_NewIntObj**, varstrings with **Tcl\_NewStringObj**, etc.

**\*\_get\_fieldobj** works like \*\_get except the field name is contained in the passed-in field object and looked up.

**\*\_lappend\_fieldobj** and **\*\_lappend\_field\_and\_nameobj** append the specified field from the pointed-to row and append the field name (via a continually reused name object) and value, respectively.

**\*\_lappend\_nonnull\_field\_and\_nameobj** works just like \*\_lappend\_field\_and\_nameobj except that it doesn't append anything when the specified field in the pointed-to row is null.

**\*\_get\_string** - This is particularly useful for the C coder. It takes a pointer to an instance of the structure, a field number, a pointer to an integer, and a pointer to a Tcl object and returns a string representation of the requested field. The Tcl object is used for certain conversions and hence can be considered a reusable utility object. The length of the string returned is set into the pointed-to integer.

Example:

```
CONST char *cable_info_get_string (struct cable_info *cable_info_ptr, int field, int *lengthPtr, Tcl_Obj *utilityObj) {...}
```

For fixed strings and varstrings, no copying is performed -- a pointer to the row's string is returned. Hence they must be considered to be constants by any of your code that retrieves them.

**\*\_delete\_all\_rows** - give a pointer to the StructTable for an instance, delete all the rows.

# Interfacing with Speed Tables From C

# 12

Interfacing with Speed Tables From C

This chapter explains the current state of interfacing with Speed Tables directly from C. You do not need to read this section unless you're interested in doing that.

At the time of this writing, no C code has been written to use any of these routines that is not part of the Speed Table code itself.

We envision providing a way to write C code inline within the Speed Table definition and, for more complicated code writing, to provide a way to compile and link your C code with the generated C code.

In particular, generating search compare functions in native C, where you say something like

```
if (row->severity > 90 && row->timeUnavailable > 900) return 1;
```

...and that gets compiled into a specifically invocable search that will be faster than our more general searches that aren't pre-compiled.

This will require generating an include file containing the structure definition, function definitions for the C routines you'd be calling, and many other things currently going straight into the C code. These changes are fairly straightforward, however, and are on the "to do" list.

# Troubleshooting

# 13

Troubleshooting

This chapter describes what to do when speed tables doesn't work.

## Compiler Errors and Warnings When Building a Speed Table

Speed Tables has been carefully coded to generate C code that will compile cleanly, specifically with the GNU C Compiler, gcc 3.3 and gcc 4.0. Right now we run the compiler with error warning levels set very high and any warnings causing the speed tables library generation process to fail. This has helped us to catch many bugs during development and we've done the work to make sure all the routines are being used with correct argument types, etc.

Should you come across a compiler warning that stops the speed table generation process, you may want to look at speed tables' software and try to fix it.

If you want to see what compiler commands speed tables is executing, you can turn on compiler debugging.

```
set ::ctable::showCompilerCommands 1
```

Do this after your "package require speedtable" and before you declare your C extensions.

How we invoke the compiler can be found in **gentable.tcl**. We currently only support FreeBSD and Mac OS X, and a general solution will likely involve producing a GNU configure.in script and running autoconf, configure, etc. We'd love some help on this.

## Simple Syntax Errors May Cause Compiler Errors

Most syntax errors in a C extension definition will be caught by speed tables and reported. When sourcing a speed table definition, you may get the message

```
(run ::ctable::get_error_info to see speed table's internal errorInfo)
```

This means that speed tables has caught some kind of unexpected internal error within itself. It has suppressed its own error traceback because it isn't valuable to anyone who isn't looking to dig into the error.

If you're not running tclsh interactively, you'll probably want to do so and then source in whatever is causing the error. After you get the above error message, you can execute...

```
::speedtable::get_error_info
```

...to see what the commotion is about.

A known bug in early December of 2006 is that if you define two fields in a table with the exact same name, you'll get a semi-strange traceback rather than a nice message telling you what you did. That's kind of characteristic of what I'm talking about.

## Core Dumps at Runtime

Speed Tables shouldn't ever dump core but, if it does, you may want to try to figure it out. If you want to be able to use your C debugger on the speed tables code, turn on compiler debugging after you've loaded the speedtable package and before you load your extension.

```
set ::ctable::genCompilerDebug 1
```

Ideally you'll also build Tcl with debugging enabled. When building Tcl, add `--enable-symbols` to your configure options to get a Tcl library that you can run your debugger over.

Run gdb on tclsh and when you hit your segmentation violation or whatever, if all is well, gdb should be on the line where the trap occurred and let you poke around all of the C variables and structures and the like.

If gdb can't find any symbols, try moving up through some stack frames (see gdb's documentation for more information). If in the speed tables routines you aren't getting file name and line number information and access to your local variables and the like, you probably haven't managed to build it with debugging enabled. Turn on showing compiler commands and make sure you see **-g** being specified when the commands are being run.

If you don't see the compiler being run, try deleting the contents of your build directory. That'll trigger a regeneration and recompile of the speed table code for your extension.



This chapter contains numerous examples of using speed tables.

### Copy Speed Table Search Results to a Tab-Separated File

```
tableType create t
...
set fp [open t.out w]
t search -write_tabsep $fp
close $fp
```

This copies the entire table `t` to the file `t.out`. Note that you could as easily have specified an open socket or any other sort of Tcl channel that might exist in place of the file. You could restrict what gets copied using addition search options like `-compare {{> severity 90}}` `-fields {name device severity}`.

### Load a Speed Table using `read_tabsep`

```
tableType create t
set fp [open t.out r]
t read_tabsep $fp
close $fp
```

### Using Copy In For Super Fast Speed Table-to-PostgreSQL Transfers

Here's the PostgreSQL syntax for copying from a file (or stdin) to a table:

```
COPY tablename [ ( column [, ...] ) ]
FROM { 'filename' | STDIN }
[ [ WITH ]
  [ BINARY ]
  [ OIDS ]
  [ DELIMITER [ AS ] 'delimiter' ]
  [ NULL [ AS ] 'null string' ]
```

```

[ CSV [ HEADER ]
    [ QUOTE [ AS ] 'quote' ]
    [ ESCAPE [ AS ] 'escape' ]
    [ FORCE NOT NULL column [, ...] ]

```

Here's an example of taking a speed table and copying it to a PostgreSQL table.

```

package require Pgtcl

source cpescan.ct

package require Cpe_scan

cpe_scan null_value \\N
cpe_scan create cpe

set fp [open junk]

cpe read_tabsep $fp

close $fp

puts [cpe count]

set db [pg_connect www]

#
# note double-backslashing on the null value and that we set the null value
# to match the null_value set with the speed table.
#
set res [pg_exec $db "copy kfoo from stdin with delimiter as '\t' null as '\\\\N"]

#
# after you've started it, you expect the postgres response handle's status
# to be PGRES_COPY_IN
#
if {[pg_result $res -status] != "PGRES_COPY_IN"} {
    puts "[pg_result $res -status] - bailing"
    puts "[pg_result $res -error]"
    exit
}

#
# next you use the write_tabsep method of the speed table to write
# TO THE DATABASE HANDLE
#
#cpe write_tabsep $db ip_address status ubr
cpe write_tabsep $db

#
# then send a special EOF sequence.
#
puts $db "\\."

```

```
#  
# the result handle previously returned will now have magically changed  
# its status to the normal PGRES_COMMAND_OK response.  
#  
puts [pg_result $res -status]
```

NOTE that all the records must be accepted by PostgreSQL, i.e. not violate any constraints, etc, or none of them will be.

Karl Lehenbauer

7/19/06 off-and-on through 1/07 and counting...

(last edit March 11, 2008 2:38 PM)

#### Acknowledgements

I would like to acknowledge Peter da Silva, the first outside user of speed tables and the first person to use them in real production code with millions of rows of data.

His insight and experience greatly contribute to the design and evolution of the technology. In particular, thanks to him it is easier to use, more capable, faster, and more memory-efficient.

He's contributed a lot on the client/server side, making it possible to "fail over" to a new server-side speed table definition in a very transparent way.

In addition, the query optimizer was completely his idea and implementation.