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

2

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. [1]

    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.

1 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 devices 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 3

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

The following data types are available[2]:

- *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

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

Example Speed Table Definition 4

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 it into a shared library.
- Multiple speed tables can be defined in one CExtension definition.
- No matter how you capitalize it, the package name with 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”.[3] The name and show fields in the row are set to the passed-in values.

3 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} \
          -include_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*_SafeInit 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.

Where Stuff Is Built

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*

For the examples, assume we have done a "`cable_info create x`"

- *set*

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

or

```
x set key 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]
```

- *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
```

- *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
```

- *search*

  Search for matching rows and take actions on them, with optional sorting.

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.

Brute-Force Search Is Fast

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

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

```
$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?
?-array_get_with_nulls varName? ?-include_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".

- `-include_field_names 1`

  If you are doing -write_tabsep, `-include_field_names 1` will cause the first line emitted to be a tab-separated list of field names.

- `-key keyVar`
- `-get listVar`
- `-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 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.

Note it is a common bug to use `-array_get` in a `-code` loop and not *unset* the array before resuming the loop, resulting in null variables not being unset, that is, in the previous row fetch, field x has 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 or use array_get_with_nulls.

- `-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 is false.

  - `{true field}`

    Expression compares true if field is true.

  - `{null field}`

    Expression compares true if field is null.

  - `{notnull field}`

    Expression compares true if field is not null.

  - `{< field value}`

Expression compares true if field less than value. ...works with both strings and numbers, and yes, compares the numbers and 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.

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:

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

Run some code every everything in the table matching above:

```
$speed table search \
    -compare {{> coolness 50} {> hipness 50}} \
     -key key -array_get data -code {
         puts "key -> $key, data -> $data"
    }
```

- *search+*

Search for matching rows and take actions on them, exploiting skip lists (indexes on a field), with optional sorting. search+ is a bit of a hack inso-far as it should be folded into search.

```
$speed table search+ ?-sort {?-?field..}? ?-fields
fieldList? ?-compare list? ?-countOnly 0|1? ?-key key-
Var? ?-offset offset? ?-limit limit? ?-code codeBody?
?-write_tabsep channel? ?-include_field_names 0|1?
```

Skip lists have significantly higher insert overhead -- it takes about 7 mi-croseconds per row inserting a million two-varchar field rows, including allocating, adding the hashtable entry, and adding the skip list node, ver-sus about 2.3 microseconds per row just to allocate the space and do the hashtable insert without the skip list insert.

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.

The one huge win of skip lists is the "range" comparison method. We are seeing speedups of 200-300X when a "range" can be used on an indexed field versus search's brute force scanning.

- *foreach*

DEPRECATED (use "search" instead)

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.

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

If you want to do something with the keys, like access data in the row, use get, array_get or array_get_with_nulls, etc, within the code body.

```
% x foreach key {
    catch {unset data}
    array set data [x array_get $key]
    puts "$key:"
    parray data
    puts ""
  }
```

`x foreach varName ?pattern? codeBody` - an optional match pattern in "string match" format will restrict what is presented to the code body.

The normal Tcl semantics for loops are followed; that is, you can execute "continue" and "break" to resume the code with the next row and break out of the foreach loop, respectively.

- *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 pgTclResultHandle
  ```

  Given a *Pgtcl* result handle, *import_postgresql_result* will iterate over all of the result rows and create corresponding rows in the table.

  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.

- *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"
```

- *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?
?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 whether you like it or not, unless if -nokeys is specified, the key value is not written to the destination.

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.

- *read_tabsep*

```
x read_tabsep channel ?-glob pattern? ?-nokeys?
?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 `-nokeys` is specified, the first field of each row is not used as the key -- rather, the key is automatically created as an ascending integer starting from 0.

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, well, there isn't currently a mechanism for finding out what the next auto key value is or to increment it. Perhaps something like "#auto" for the key value or a -nokey argument to the set method would be a nice addition.

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

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

Currently brute force searching (searches visiting all rows in a table) is best performed with *search* rather than *search+*, as search is significantly faster at visit-every-row searching.

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 110 nanoseconds per row or searching almost ten million rows per CPU second on circa-2006 AMD64 machines, without doing a lot of work to optimize the performance of the code.

If you need to search for ranges of things, partial matches, etc, you can use indexes in conjunction with *search+* and the range compare function 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 `$speed table index create $fieldName`. Currently only the range comparison function is accelerated, and only while using search+. In addition, the range function must be the first element of the list of comparison functions specified by the `-compare` option to the search method.

search+ is likely to go away in the future as its capabilities will be absorbed into a more powerful version of *search* that has yet to be written.

Client-Server Speed Tables 7

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 used shared memory and supported multiple readers and writers, as processes and/or threads, with locking, etc, would be great. We don’t currently support that and, while it wouldn’t be that super hard to get going, it’s one of those deals where it’s hard to get 100% right and in the meantime you’re going to endure bugs or worse. [4]

Even with this limitation, as they currently exist, 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 implementation provides 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*.

Authentication, lack thereof

4 In particular the skip list routines have all sorts of stuff local to themselves that makes them single-user until reworked.

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[5], 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’s large enough, you may start considering having machines basically serve as dedicated Speed Table servers. Stuff generic machines 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.

Example client code:

```
package require speed table_client

remote_speed table speed table://127.0.0.1/dumbData t

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

5 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.

C Code Generated and C Routines Created

8

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.)

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 *speedTableCreatorTable* associated with it, for example:

```
struct speedTableCreatorTable {
    Tcl_HashTable     *registeredProspeed tablePtr;
    long unsigned int  nextAutoCounter;

    int                nFields;
    int                nLinkedLists;

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

    struct speed tableFieldInfo **fields;

    void *(*make_empty_row) ();
    int (*set) (Tcl_Interp *interp, struct speed tableTable *speed table, Tcl_Obj *da-
taObj
, void *row, int field, int indexCtl);
    int (*set_null) (Tcl_Interp *interp, struct speed tableTable *speed table, 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 speed tableSearchStruct *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 *speed tableFieldInfo 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 speed tableTable {
    struct speed tableCreatorTable      *creatorTable;
    Tcl_HashTable                       *keyTablePtr;
    Tcl_Command                          commandInfo;
    long                                 count;
    jsw_skip_t                         **skipLists;
    struct speed table_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 speed table_types cable_info_types[] = {
    Speed Table_TYPE_INET,
    Speed Table_TYPE_MAC,
    Speed Table_TYPE_VARSTRING,
    Speed Table_TYPE_VARSTRING,
    Speed Table_TYPE_VARSTRING,
    Speed Table_TYPE_VARSTRING,
    Speed Table_TYPE_INT,
    Speed Table_TYPE_INT,
    Speed Table_TYPE_LONG,
    Speed Table_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 speed table_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_nonuull_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_nonull_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

9

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 invokable 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 10

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 ::speedtable::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 ::speedtable::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 ::speedtable::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.

Examples

11

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 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 12/15/06 and counting...

(last edit December 25, 2006 10:09 PM)