Speed Tables - A High-Performance, Memory-Resident Database for Tcl

Karl Lehenbauer
Peter da Silva

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 a table definition and generates C code to create and manage corresponding structures, 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, search and/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.

In contrast to ad-hoc tables implemented with some combination of arrays, lists, upvar, namespaces, or even using Tcl 8.5's dicts, speed tables' memory footprint is far smaller and performance far higher when many rows are present.

Speed tables' search capabilities include indexed searches, results sorting, setting offsets and limits, specifying match expressions, and counting. A configurable searching engine, speed table searches bypass the Tcl interpreter on a row-by-row basis (except for processing matches), providing high performance. 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, again bypassing the interpreter on a per-row basis.

Representing Complex Data Structures in Tcl

Tcl is not 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, it is possible to 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 at least 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.

   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.

   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 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, fields are stored in native formats such as short integer, integer, float, double, bit, etc, and field names only occur once for a table type regardless of the number of tables created and the number of rows in those tables.

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

---

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.)

on the rows that match your search criteria.

Null Values

Speed tables maintains a "null value" bit per field, unless told not to, and provides 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.

Indexes

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 many of optimizations in place to make speed tables fast.

Speed Table Data Types

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
- mac - an ethernet MAC address
- *inet* - an internet IP address
- *tclobj* - a Tcl object

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

```
double longitude
```

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

Configurable Field Attributes

Field definitions can 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” 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 sorting (since skip lists are sorted). This defaults to “indexed 0”, i.e. the field is not generated with index support.

- *notnull*

If notnull is specified as true, the code generated for the speed table will not have code for maintaining an out-of-band null/not-null status created for it, increasing the performance of ma-

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

nipulating fields for which out-of-band null support is not needed. Defaults to “notnull 0”.

• *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 notnull, 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.

• *key*

If key is specified as true, this field will become the key for the table. There must be at most one “key” field, and it currently must be a varstring. (Any field type can be indexed but our Tcl-adapted hashtables require strings as indexes.) If no “key” field exists then the key will not be exposed as part of a row unless it is explicitly referenced with the name “_key”.

Special Fields

Named fields may not begin with an underscore (as these are reserved for speed table internals), but there are two special field names that may be used in any place where a field is specified:

• *_key*

If no field is specified as a key, this name can be used to reference the key for the row.

• *_dirty*

This is set whenever a field is modified, and may be explicitly cleared… for example when a table is saved to a TSV file in a search operation.

Example Speed Table Definition

```
package require speedtable

CExtension animinfo 1.1 {

      CTable 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 along with support code 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.

It is not necessary to re-execute the CExtension definition to use it again, but it is always safe (and efficient) to do so -- 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.

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 use **`set obj [animation_characters create #auto]`** to create a new instance of the table, without having to generate a unique name for it.

Basic Examples

All rows in a speed table have a unique key value, which normally resides outside of the table definition itself. The simplest way to create or modify a row is with the *set* operation. Performing a *set* on a speed table performs an update or insert:

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

If there is no row in the table with "shake" as a key [3], this creates a new row in the speed table **t**, otherwise it updates the current value of the row having the key "shake" with a new name and show.

We can set other fields in the same row:

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

And increment them in one operation with "incr":

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

You can fetch a single value naturally with "get"...

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

Or can get all the fields in the row, in the order they were defined in the ctable definition:

```
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 key-value list of fields, suitable for passing to **`array set`**, using array_get:

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

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 need to check for its existence using `info exists` before trying to use it or use *array_get_with_nulls*, which will always provide all the fields' values, substituting a null value string, and typically the empty string) when the value is null.

You can check if a key exists with *exists*:

```
t exists frylock
0
```

Or load a complete table from a file tab-separated data with *read_tabsep*:

```
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}
```

3 The key for a row has a name, "_key", but it's not exposed implicitly in operations on the default list of fields. It is also possible to use the "key" attribute to make any single varstring

```
        {> age 20}
    } \
    -array data -code {
    parray data
    puts ""
}
```

Additional meta table methods

- *animation_characters null_value \\N* - which sets the default null value for all tables of this table type to, in this case, \N.
- *animation_characters method foo bar* - this will register a new method named *foo*, which will be available to all instances of the table. Invoking the foo method will cause the *bar* proc to be called with the arguments being the name of the table 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 "*t a b c d*".

Where the table Is Built

The generated C source code, some copied .c and .h files, the compiled .o object file, and shared library are normally 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 ctable" and outside of and prior to the CExtension definition, if you invoke

```
CTableBuildPath /tmp
```

...then those files will be generated in the **`/tmp`** directory.

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

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 **`cable_info create x.`**

• *set*

There are two ways to specify the fields to set:

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

or

```
x set key keyValueList
```

The key is unique. It can be any string and is not normally a field of the table. The following commands are equivalent:

```
x set peter ip 127.0.0.1 \
    name "Peter da Silva" i 501
x set peter {
    ip 127.0.0.1
    name "Peter da Silva"
    i 501
}
```

Thus a natural way to pull an array into a speed table row is:

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

• *fields*

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

• *field*

"field" returns information about the field attributes. Since we ignore attributes we don't recognize, you can include your own key-value pairs and access them using this method: **`field getprop name`** returns the value of the *name* attribute**`. field properties`** returns a list of all attributes. **`field proplist`** will return the names and values of all the properties in the usual name-value format.

• *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. Null fields 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, in "array get" format, including null fields.

• *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*

Returns the number of rows in the table.

• *batch*

The batch command provides an efficient way to perform a series of ctable operations. It takes a list of ctable commands (without the ctable name) and returns a list of results. Each element in the result list is a list of the index of the result, and a list of two values: the Tcl result code (for example, 0 for TCL_OK, 1 for TCL_ERROR) and the result string (result or error string).

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

Dean's age to 17 produced no result. Incrementing it returned the incre-

mented value (18), and trying to add ‘foo’ to Brock’s age produced 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 only return an error in the event of bad arguments such as an invalid “batch” list.

• *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.

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).

Search[4] has scads of options:

• `-sort sortArg`

Sort results based on the specified field or fields. To sort a field in descending order, put a dash in front of the field name.

• `-fields fieldList`

Restrict search *results*[5] to the specified fields, which (among other things) may produce a noticable performance boost..

• `-glob pattern`

Perform a glob-style comparison on the *key*, excluding the examination of rows not matching.

• `-countOnly 1`[6]

Counts matching rows but does not take any action based on the count.

• `-offset offset`

• `-limit limit`

Like the SQL “offset” and “limit” parameters, these limit the results to a section of the ctable. The results are not well-defined without -sort or -countOnly.

• `-write_tabsep channel`

Matching rows are written tab-separated to the file or socket (or postgresql database handle) "channel".

• `-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.

• `-compare list`

Perform a comparison to select rows.

---

4 Like berkeley ls.

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

6 All search parameters must have a value, so “-countOnly” requires the value “1”.

Compare expressions are specified as a list of lists. Each list consists of an operator and one or more arguments. Each expression is applied to each row in turn, and all expressions must match for the search to succeed.

Here's an example:

```
$speedTable 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.

Most expressions are fairly easy to understand:

- `{false` *`field`*`}`
- `{true` *`field`*`}`
- `{null` *`field`*`}`
- `{notnull` *`field`*`}`

Comparisons are type-sensitive:

- `{< field value}`
- `{<= field value}`
- `{= field value}`
- `{!= field value}`
- `{>= field value}`
- `{> field value}`

String matching uses “glob” operations:

- `{match field expression}`
- `{match_case field expression}`
- `{notmatch field expression}`
- `{notmatch_case field expression}`

Range and are the most efficient when performed on indexed fields:

- `{range field low hi}`

List operations must be performed on indexed fields, and only one may be in a list. Yes, this is not optimal and will be changed in a future release:

- `{in field valueList}`

- `-code codeBody`

Run scripting code on matching rows, along with one or more of these options:

- `-key keyVar`

Make the key value of the matched row be available inside the code block as *keyVar*.

- `-get listVar`

The fields of the row are available in the variable *listVar*.

- `-array arrayName`
- `-array_with_nulls arrayName`

The fields are available as the array *arrayName*.

- `-array_get listVar`
- `-array_get_with_nulls listVar`

The fields are available in an “array get” format list in *listVar*.

Search 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"
    }
```

- *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

- *key*

  Return the name of the “key” field in the ctable (usually _key).

- *makekey*

  Given a list of name-value pairs, return the key value.

- *methods*

  Return a list of defined methods (commands) that the ctable can handle. The speedtable API may include extensions (such as the ctable server) or implement ctable-compatible classes independently of ctables (for example, there’s a STAPI definition for pgsql), so it may be necessary to check whether the STAPI-compatible object that you are examining supports the commands you need.

- *store*

  x store keyval_list

  Stores the list in the ctable using the key defined in the ctable definition, or using an autoincremented numeric key compatible with read_tabsep if the table’s key field is not specified in the list.

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

```
set res [pg_exec $connection \
    "select * from mytable"]
if {[pg_result $res -status] ==
"PGRES_RESULT_OK"} {
  x import_postgres_result \
    $res
}
$res destroy
```

- *fieldtype*

  Return the data type of the named field, such as varstring.

- *needs_quoting*

  Given a field name, return 1 if it might need quoting. For example, varstrings and strings may need quoting, while integers, floats, IP addresses, MAC addresses, etc, do not.

- *names*

  Return a list of all of the keys in the table. This is fine for small tables but horribly inefficient for large tables; use *search* instead.

- *reset*

  Clear everything out of the table.

- *destroy*

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

- *read_tabsep*

  Read tab-separated entries from a Tcl 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 column is normally the key and is not included in the list of fields. So if you name five fields, for example, each row must contain six columns.

  Options:

  - *-glob pattern*

    If the key does not match, the row is not inserted.

  - *-nokeys*

    The first column is not a key column. If the table has a key field defined, and that column is in the fields being read, then it will be used. Otherwise an auto-incremented numeric key will be generated for each row and read_tabsep will return the last key generated.

    read_tabsep stops when it reaches end of file OR when it reads an empty line.

- *index*

  Index actually creates the index for fields with the indexed attribute. This is a separate operation because it is far more created the indices AFTER populating a large table.

  ```
  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. 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 count foo
  ```

  Returns a count of the skip list for field "foo".

  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.

```
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.

Performance Importing PostgreSQL Results

On a 2 GHz AMD64 running FreeBSD, speed tables can import import about 200,000 10-element rows per CPU second, i.e. around 5 microseconds per row. Importing is slower if one or more fields has an index.

Speed Table Search Performance

An example of brute force searching that there isn't much getting around without adding fancy full-text search features is unanchored text search. Even in this case, with speed tables's fast string search algorithm and quick traversal during brute-force search, the authors have observed 60 nanoseconds per row, thus 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.

Indexed Searches

Many searches can be greatly accelerated through the use of indexes on appropriate fields. Please consult the documentation for which operators and under what conditions indexes cause searches to be accelerated.

Client-Server Speed Tables

Tables created with Speed Tables are, by default, local to the Tcl interpreter that created them.

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 (requiring values in all cases) 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 *most speed table methods*.

Stored Procedures

There is a Tcl interpreter on the server side, pointing to the possibility of deploying server-side code to interact with Speed Tables[7], although there isn't any formal mechanism for creating and loading server-side code at this time.

---

7 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.

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.

Dedicated Speed Table Servers

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 want 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 that traditional SQL databases. This is a lot stronger than *memcached* because memcached is basically just a directory of files. Here, at the very least, you can define fields that contain metadata and use search to look stuff up.

Speed Table URLs

```
sttp://foo.com/bar
sttp://foo.com:2345/bar
sttp://foo.com/bar/snap
sttp://foo.com:1234/bar/snap
stty://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 table name, and optional extra stuff. Currently the optional directory and optional extra stuff are parsed, but ignored.

Example Client Code

```
package require speedtable_client

remote_speedtable \
speedtable://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

When registering a table on the server side, use a wildcard for the host:

```
package require speedtable_server
# create a ctable 't' here.
::speedtable_server::register \
speedtable://*/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.

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, batching and using searches in places of gets can have a dramatic

impact on client/server speed table performance.

Batching

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 (this is not checked, though, and the results are undefined). In particular, get, delete, and exists can be pretty useful.

Use "search" Instead of "get"

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

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

In the above example, every "get" causes 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_get data {
    ....
}
```

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 behave 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 loop.

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

Speed tables can use shared memory to accelerate concurrent access by multiple processes. The design objective was to provide a way for same-server clients to access the speed ta-

ble through shared memory while retaining the ability to build and use speed tables without using shared memory at all.

When a speed table is instantiated for use with shared memory, the entire table, all keys and indexes are stored in shared memory, and may be used when there is sufficient memory available.

Tricky synchronization issues surfaced quickly while development this. 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 be permitted to disappear behind search's back.

Also search tables were 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. In many cases all changes are handled by a single process anyway, and no updates need be sent from clients.

We take advantage of the skiplist code's ability to support lockless synchronization between processes sharing 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 number 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.

To use shared memory support, a new parameter was added to the "create" command, specifying that a table was shared as a *master* or a *reader*. This parameter is followed by a list of options that describe the size of the table and how it is built.

STAPI - the Speed Table API

STAPI creates the speedtables API, which is used for a variety of table-like

objects. This includes remote speed tables 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 speed table from an SQL table as a local 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 read-only cache for a remote database for many client processes.
- *st_client*, which provides the general interface for creating STAPI objects identified by URIs.

  The primary mechanism for using STAPI as a client is through **::stapi::connect**, which connects to a speed table server or other database providing a speed table interface via a URI.

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

Only one option is normally required:

```
-key  col
```

Define the column used to generate the key.

If a key is not 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 have been defined in STAPI:

Using a ctable server via sttp (client/server)

`package require st_client`

sttp://[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 the client-server speed table transfer protocol for all other methods.

An additional option is used:

```
-build directory
```

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_pgtcl`

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

Create a stapi interface directly to a PostgreSQL table

The connection part has not been implemented yet. It will be something like **[user[:password]]@[host:]database**

If no columns are listed, all columns of the table will be returned.

Parameters are name=value style, just like in HTTP. A key column should be defined. Pseudo-columns can be defined here, too, with parameters like **`-column=sql_code`** or **`_key=column_name.`**

Using an already opened speed table

`package require st_client`

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.

STDisplay - Display Functions for the web

STDisplay is derived from Rivet's DIODisplay, and the calling sequence is similar. For example:

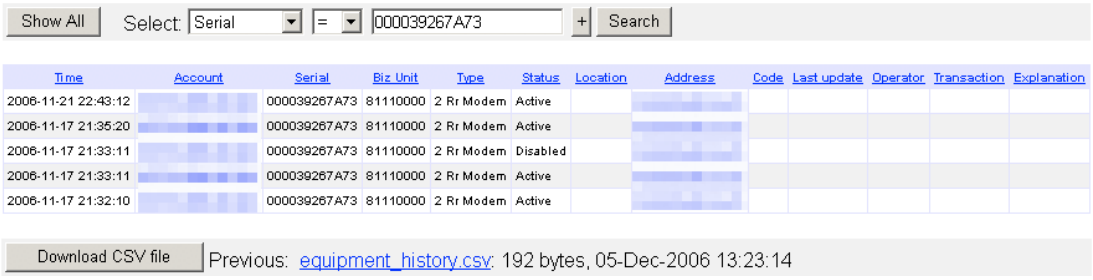
```
set display [
 ::STDisplay #auto -uri \
  sql:///history?_key=time
]

$display field time
$display field account
$display field serial
…
$display field explanation
$display show
```

Since STDisplay works with anything that can be exposed in the Speed Table API, it's an efficient and convenient mechanism to browse many kinds of database and database-like tables on the web.

Speedtable C-level implementation

You can interact with any speed table, regardless of its composition, from C, by making standardized calls via the speed table's methods (pointers to functions) and speed table's creator table structures.

Varstrings are *char* * pointers and a length. We allocate the space for whatever string is stored and store the address of that allocated space. We avoid malloc/frees when possible by reusing the space when values change and the string being store fits. Default values are represented with a



null pointer., while 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.

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 really appreciate some help on this.

License

The Speed Tables package is distributed under the same permissive Berkeley license that Tcl uses.

Obtaining Speed Tables

A SourceForge project has been requested for speed tables and should be available by September 20th, 2007.

http://sourceforge.net/projects/speedtables

Included with speed tables is a 65 page developer's manual. A test suite includes dozens of tests.

Request For Participation

Speed tables are working well, for the authors as least. An autoconf-style configuration system would be of great value. We'd also like a mechanism for defining C-based search comparison routines, and someone to use and construct examples for interfacing to C directly.

Speed tables could be a useful addition to other scripting languages as well, and of course additional documentation would always oblige, as well as manual translations into other langauges, testing for different locales, and so forth.

Summary

Speed tables powerfully extends Tcl's ability to access and manipulate data, providing unique capabilities, far higher performance, and greater memory density over traditional Tcl-based approaches. Its permissive license makes it feasible for use in a wide range of projects that need higher performance storage and searching than available from a SQL database or from ad hoc methods.