10 years of Speed Tables

Peter da Silva

Tcl 2016

October 24, 2016

What's new and what's cool

Speed Tables is an in-memory key-value store with a non-SQL single-table database query language, along with a set of front ends (the Speed Tables API) that emulate Speed Tables with a variety of back-ends including client-server connections to remote tables, PostgreSQL, and most recently Cassandra.

This talk will provide a brief overview of Speed Tables and discuss the changes and improvements since 2007.

What are Speed Tables?

Speed Tables are a key-value store that serves the dual purpose of providing a more compact and higher performance alternative to Tcl hashes for heterogenous data, and providing a standard API to structured searchable tables. They are very fast because they are implemented in custom Tcl extension in C that is generated on the fly and compiled by taking a Speed Table definition and generating a custom Tcl extension that implements one or more Speed Tables in a Tcl extension that can then be loaded and used like any other extension:

```
CExtension particles 1.0 {
  CTable quark {
    key id
    double charge indexed 1 notnull 1 default 0.0
    varstring color indexed 1 notnull 1 default red
    varstring flavor indexed 1 notnull 1 default top
  }
}
```

This creates a Tcl extension called "Particles" that can now be loaded and used:

```
package require Particles
quark create accelerator
accelerator index create color
accelerator index create flavor
accelerator store id q0001 charge 0.3333 color red flavor strange
# ...
accelerator store id q6666 charge -0.6777 color green flavor top
accelerator search \
    -compare {{= color red} {= flavor strange}} \
    -array row -code {
        puts "Found a red strange quark id = $row(id)"
    }
```

Searches on the key field, or indexed fields, are blindingly fast. The search command re-orders the query to perform the fastest and most efficient search first, and uses the rest as filters on the result. Indexed searches can be also be performed on shared-memory tables with no locking required, making the shared memory tables super-efficient.

What's new in Speed Tables?

The original implementation of Speed Tables was pure C, with manual memory management, and assumed a 32-bit memory model. It's been extended to support 64-bit architectures, and has taken advantage of a lot of components in the C++ Boost library.

To speed up searches and to extend the simple key-value store to provide an analogue of database views, inline C filters have been added. These are defined with C fragments embedded in the high level table definition, which are wrapped and inserted inline into the generated C code and can be used to perform complex low-level operations during searches.

A lockless shared memory implementation with a single master and multiple read-only slaves has been put into production at FlightAware. Performance is excellent, within a few percent of non-shared Speed Tables even for the master, because writes are never locked.

The core Speed Tables methods and search syntax have turned out to be a good general-purpose interface to indexed key-value data stores. A common implementation of the API in Tcl with multiple back-ends has allowed systems to migrate pretty much transparently between native C Speed Tables, a client-server API using network sockets, and PostgreSQL. Despite exposing some differences between back-ends, it's possible to run the same code with local or remote tables, using C tables or PostgreSQL.

The latest addition to the Speed Tables API is a Cassandra front end that converts each high level search component in the Speed Tables request into either CQL or Tcl code, based on the capabilities of the underlying Cassandra schema.

Query Optimizer

The original search operation simply walked the index and kicked off a callback on each matched row. Adding indexes allowed a much faster search, but it was still necessary to make sure that an indexed field was the first item in the row. I first implemented a "query optimizer" in Tcl that looked for indexed fields and would have to be manually included in the call:

```
table search -compare [optimize {= field value} {< field value} ...]
```

This was cumbersome, and so I re-implemented it internally in the search function, and modified it to assign scores to different types of comparison operator. The scoring was tweaked a little based on the searches I was running into when this was first implemented, but it hasn't really been revisited in several years. The base score table, indexed by the internal code for the search type, looks like this:

```
{SKIP_START_NONE,     SKIP_END_NONE,    SKIP_NEXT_NONE, -2 }, // FALSE
{SKIP_START_NONE,     SKIP_END_NONE,    SKIP_NEXT_NONE, -2 }, // TRUE
{SKIP_START_NONE,     SKIP_END_NONE,    SKIP_NEXT_NONE, -2 }, // NULL
{SKIP_START_NONE,     SKIP_END_NONE,    SKIP_NEXT_NONE, -2 }, // NOTNULL
{SKIP_START_RESET,    SKIP_END_GE_ROW1, SKIP_NEXT_ROW,   2 }, // LT
{SKIP_START_RESET,    SKIP_END_GT_ROW1, SKIP_NEXT_ROW,   2 }, // LE
{SKIP_START_EQ_ROW1,  SKIP_END_GT_ROW1, SKIP_NEXT_ROW,   6 }, // EQ
{SKIP_START_NONE,     SKIP_END_NONE,    SKIP_NEXT_NONE, -2 }, // NE
{SKIP_START_GE_ROW1,  SKIP_END_NONE,    SKIP_NEXT_ROW,   2 }, // GE
{SKIP_START_GT_ROW1,  SKIP_END_NONE,    SKIP_NEXT_ROW,   1 }, // GT
{SKIP_START_NONE,     SKIP_END_NONE,    SKIP_NEXT_NONE, -2 }, // MATCH
{SKIP_START_NONE,     SKIP_END_NONE,    SKIP_NEXT_NONE, -2 }, // NOTMATCH
{SKIP_START_GE_ROW1,  SKIP_END_GE_ROW2, SKIP_NEXT_MATCH, 3 }, // MATCH_CASE
{SKIP_START_NONE,     SKIP_END_NONE,    SKIP_NEXT_NONE, -2 }, // NOTMATCH_CASE
{SKIP_START_GE_ROW1,  SKIP_END_GE_ROW2, SKIP_NEXT_ROW,   4 }, // RANGE
{SKIP_START_RESET,    SKIP_END_NONE, SKIP_NEXT_IN_LIST,  6 }  // IN
```

There are additional modifiers that don't show up in this table, for example there's a bonus for being the field the search operation is sorted on, because the index is ordered and thus the sorting step can be avoided if a the right index is used. If the key field is being matched for equality, then it gets an infinite bonus because the hash operation on the key is faster than walking the index for large tables. And the case-sensitive match operation only gets a bonus if it's an anchored match that allows the first part of the match to be pulled from the index.

There are some situations where the optimizer is not going to do a good job, where the programmer could beat it based on information about the actual content of the table that the optimizer doesn't have available. For example "non NULL" should have a higher score if you know that particular index is sparse, and there are obviously going to be cases where the hash is beaten by an index that happens to be a tighter constraint for that particular search, but right now there's no way to tell search to override the optimizer. You can tell what field you prefer with the "`-index`" option, but if it thinks a different field is better it will use it.

CFilters - Inline C in Speed Tables

The "compare" operation in search is basically like a series of "WHERE test AND test AND test..." operations, but sometimes you want to perform more complex comparisons. Rather than create a complex query language, that would be interpreted and slow down Speed Tables blindingly fast searching, it was decided to add the ability to add the ability to embed filters in the table definition itself. Since the table was implemented in C, the filters are themselves tiny fragments of C code. Filters are evaluated using a "-filter" option on the search command that is evaluated after any other comparisons, and can be applied to any search whether indexed or not.

Here is a very simple filter in a "luggage" table:

```
CExtension Filtertest 1.0 {
  CTable luggage {
    key id
    varstring owner indexed 1
    double x notnull 1 default 0.0
    double y notnull 1 default 0.0
    double z notnull 1 default 0.0

    cfilter oversize args {double lim} code {
        if(row->x > lim || row->y > lim || row->z > lim)
            return TCL_OK;
        return TCL_CONTINUE;
    }
  }
}
```

This filter could be used like so, to calculate a surcharge for each oversized parcel on a flight that has an assigned owner:

```
# calculate surcharges for oversize luggage
luggage search \
    -compare { {notnull owner} } \
    -filter { {oversize 0.6} } \
    -array row -code {
        incr oversize($row(owner)) 1
    }
foreach passenger [array names oversize] {
    set surcharge($passenger) [expr {$charge * $oversize($passenger)}]
}
```

The generated code for this function parses the arguments only once for each search - a unique sequence number is generated for each search, so the filter arguments are only parsed once no matter how may rows are matched:

```
int luggage_filter_oversize (
    Tcl_Interp *interp,
    struct CTable *ctable,
    ctable_BaseRow *vRow,
    Tcl_Obj *filter,
    int sequence)
{
    struct luggage *row = (struct luggage*)vRow;
    static int lastSequence = 0;
    static double lim = 0.0;

    if (sequence != lastSequence) {
        lastSequence = sequence;
        if(Tcl_GetDoubleFromObj (interp, filter, &lim) != TCL_OK)
            return TCL_ERROR;
    }

    if(row->x > lim || row->y > lim || row->z > lim)
        return TCL_OK;
    return TCL_CONTINUE;
}
```

Since the each row is stored as a single C++ structure, the resulting code is as fast as native C. Since the owner field is indexed, the search operation simply walks that index, rows with no owners are not even examined.

```
struct luggage : public ctable_BaseRow {
    ctable_LinkedListNode _ll_nodes[LUGGAGE_NLINKED_LISTS];
    char                 *id;
    int                   _idLength;
    int                   _idAllocatedLength;
    char                 *owner;
    int                   _ownerLength;
    int                   _ownerAllocatedLength;
    double                x;
    double                y;
    double                z;
    unsigned int          _dirty:1;
};
```

For more complex filters that are passed multiple arguments, a more complex function is generated, and with the "memoized" parameter passing the operation is as fast as possible. All the list handling and error checking is generated automatically, making the filter itself simple and easy to read. For example:

```
cfilter distance2 args {double target_lat double target_long double target_range} code {
    double dlat = target_lat - row->latitude;
    double dlong = target_long - row->longitude;
    double dsquared = (dlat * dlat) + (dlong * dlong);
    if(dsquared <= (target_range * target_range)) return TCL_OK;
    return TCL_CONTINUE;
}
```

Generates:

```
int track_filter_distance2 (Tcl_Interp *interp, struct CTable *ctable, ctable_BaseRow *vRow,
Tcl_Obj *filter, int sequence)
{
    struct track *row = (struct track*)vRow;
    static int lastSequence = 0;
    static double target_lat = 0.0;
    static double target_long = 0.0;
    static double target_range = 0.0;

    if (sequence != lastSequence) {
        lastSequence = sequence;
        Tcl_Obj **filterList;
        int       filterCount;

        if (Tcl_ListObjGetElements(interp, filter, &filterCount, &filterList) != TCL_OK)
             return TCL_ERROR;

        if (filterCount != 3) {
            Tcl_WrongNumArgs (interp, 0, NULL, "filter requires 3 arguments: target_lat,
target_long, target_range");
            return TCL_ERROR;
        }

        if(Tcl_GetDoubleFromObj (interp, filterList[0], &target_lat) != TCL_OK)
            return TCL_ERROR;
        if(Tcl_GetDoubleFromObj (interp, filterList[1], &target_long) != TCL_OK)
            return TCL_ERROR;
        if(Tcl_GetDoubleFromObj (interp, filterList[2], &target_range) != TCL_OK)
            return TCL_ERROR;
      }

      double dlat = target_lat - row->latitude;
      double dlong = target_long - row->longitude;
      double dsquared = (dlat * dlat) + (dlong * dlong);
      if(dsquared <= (target_range * target_range)) return TCL_OK;
      return TCL_CONTINUE;

}
```

Fast data imports and exports

To quickly populate Speed Tables, a number of specialized methods have been created. Initially, Speed Tables could be populated via flat text files with tab-separated values (TSV files). This was extended to importing PostgreSQL results via the pgtcl library, and later Cassandra results from casstcl. For example:

```
set r [pg_exec $conn "select id, name, type, weight from animals;"]
if {"[pg_result $r status]" == "PGRES_COMMAND_OK"} {
    $table import_postgres_result $r
}
pg_result $r -clear
```

Shared Memory Speed Tables

A Speedtable and its indexes can be placed in a shared memory segment, and accessed (read only) by multiple processes through the search command on the indexed fields. This allows a single master program to update an in-memory datastore that's used by multiple child processes.

The use of skiplists means that no locking is required in this scenario. Since Skiplists are allocated probabilistically and never rebalanced, it's possible to add and remove elements from the list such that the list is always valid, even if a member is removed while another process is walking it, so long as no list member or any of its memory is deallocated until its known to be unreferenced.

To maintain this constraint, a clock is used. The master updates the clock each time it allocates or deallocated memory. Each reader asks the master for a word in shared memory that it can update, and when it starts a search operation it copies the current clock into its shared memory. When the master releases memory, it copies the current clock to it and only actually deallocates it when its clock is older (using modular arithmetic) than the clock of *any* reader.

The table is initially created in the master process, via

```
speedtable create table master {file filename size bytes ...}
```

Each reader needs to communicate with the master process to request a slot, which is returned as a Tcl list containing the parameters passed to master, and a token letting the reader know how to connect to the table and access its slot. This communication happens outside the shared memory speed tables API, perhaps via a local socket connection. The client passes its pid to the master, the master calls `table attach pid` and passes the list returned back to the client that then creates a matching "reader table", via

```
speedtable create table reader $list
```

The reader can only perform a limited number of Speed Table commands, primarily search, and certain options of search like `-delete` are disallowed.

Speed Tables API

The shared memory speed tables were developed together with a client-server API, which provided a socket-based wrapper in Tcl that allowed a program to connect to a remote process and access its Speed Tables directly. Instead of creating a table internally, a client would connect with:

```
remote_ctable ctable://host:port/name table
```

This would connect to the specified host and port and pass the provided name though, similar to the way most command-based TCP protocols, like HTTP, operate. Then the server would accept forwarded Speed Table commands and respond with the results.

This allowed full read-write access to the remote Speed Table:

```
% remote_ctable ctable://localhost:161/m m
% m count
1200
% m store [list id 1001 owner joe x 1.0 y 1.0 z 0.2]
% m array_get 1001
id 1001 owner joe x 1.0 y 1.0 z 0.2
```

Performance is somewhat impacted by the extra reads and writes, and the server can only support so many readers, but for one-to-few connections and short transactions it works very well. And it turned out to be a really great way to handle the handshaking required by Shared Memory Speed Tables.

```
% set params [m attach [pid]]
% ctable create r reader $params
% r share info
size 4194304 flags {core sync shared} name m creator 1 filename
sharefile.dat base 232
% r search -compare { {= owner joe} } -array row -code {
    puts $id
}
1001
```

Now the client can use the remote Speed Table for read-write access to the table, and the shared version for super-fast searches. Still, keeping track of the two was inconvenient. So the next step was to create a generalized Speed Tables API.

```
% set remote [::stapi::connect sttp://localhost:1616/]
% $remote count
1200
```

The routine `::stapi::connect` creates a Speed Tables API object that accepts the same methods as Speed Tables. With a syntax callously ripped from HTTP, this can clearly support a variety of protocols. The first protocols implemented were `sttp://` (Speed Table Transfer Protocol, basically the client-server protocol) and `shared://` (An STTP connection and a Shared Memory Speed Table working together).

```
% set table [::stapi::connect shared://1616/master]
```

Methods that can be performed using Shared Memory Speed Tables are handled locally, and commands that require action by the server are handled by STTP.

Since then, we have added `sql://` (PostgreSQL) and `cass://` (Cassandra) all accessed via `::stapi::connect`.

PostgreSQL (sql://)

The SQL implementation will map the underlying table directly, by reading its schema out of PostgreSQL, or a more complex STAPI URL can be constructed that assembles the exposed Speed Table out of fragments of SQL. The URL should at least specify which column or group of columns will be used for the key field, but recent versions of STAPI can even extract that from the schema.

For example, suppose we have a table:

```
create table stapi_test (
       isbn   varchar primary key,
       title  varchar,
       author varchar,
       pages  integer
);
```

One would connect with:

```
set st [::stapi::connect sql:///stapi_test]
```

This would produce a command `::stapi::sqltable1::ctable` that behaves just like a speedtable instance. If queried with:

```
search -compare {{match isbn 1-56592-*}} -key k -array row {
    parray row
}
```

STAPI would convert it to the query:

```
SELECT isbn AS __key,* FROM stapi_test WHERE isbn ILIKE '1-56592-%';
```

The key is extracted separately to maximize compatibility with Speed Tables, since it's optional to name a key, though in practice the simulated table could maintain the original key behavior in most cases. The Cassandra wrapper doesn't copy this quite so slavishly, and perhaps this could be re-addressed later.

Cassandra CQL (cass://)

Cassandra has a more complex schema, with partitioning and clustering keys, but luckily it also makes it easy to extract these from the schema... so a Cassandra URL can still be as simple as `cass://host:port/tablename`, and the STAPI wrapper will extract key and index information from the schema.

So, given a Cassandra table:

```
CREATE TABLE test.class (
        room text,
        hour int,
        subject text,
        PRIMARY KEY (room, hour)
);
```

One can connect with:

```
set class [::stapi::connect cass:///test.class/]
```

And query with:

```
$class search -compare {{range hour 9 13} {in room {1301 1302}}} -array
row -code {
        parray row
}
```

Since STAPI knows the partition and clustering keys from the schema, it knows what operations are possible on which fields, and can generate this code:

```
SELECT * FROM test.class
    WHERE room IN ('1301','1302')
      AND hour >= 9 AND hour < 13;
```

For some queries, one does have to add the Cassandra-only `-allow_filtering 1` option to the search. This incompatibility with the Speed Tables API is perhaps arguable, but consider this request:

```
$class search -compare {{>= hour 12}} -array row -allow_filtering 1 -code
{
        parray row
}
```

Which would generate the following naive CQL:

```
SELECT * FROM test.class WHERE hour >= 12;
```

Since the partition key isn't present in the query, this code will fail with an error `"Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING"`. Now, in principle, the cass:// method could detect this situation and automatically insert `ALLOW FILTERING`. But this isn't good practice. Cassandra has to request *all* partitions and then filter the result for `hour >= 12`, which is obviously very expensive. An explicit request by the user to allow filtering is required:

```
$class search -compare {{>= hour 12}} -array row -allow_filtering 1 -code
{
        parray row
}
```

Which tells Cassandra you really mean to perform this expensive request:

```
SELECT * FROM test.class WHERE hour >= 12 ALLOW FILTERING;
```

Conclusion

The Speed Tables API, particularly the search operator, is an extremely clean high level interface to databases and database-like structures. The syntax is easy to parse and analyze, and map into a variety of packages that can be viewed as indexed key-value datastores. It's easy to extend, with filters, and process with tools like the query optimizer, and the underlying simple native C tables and Skip List based indexes made it possible to extend to a unique shared-memory solution.