10 years of Speed Tables

Peter da Silva
FlightAware

What are Speed Tables?

What are Speed Tables?

- An array of structures
- A Key-Value store
- A “NoSQL” database
- A portable API

Example

```
CExtension particles 1.0 {
  CTable quark {
    key id
    double mass indexed 1 notnull 1 default 0.0
    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
  }
  CTable lepton {
    key id
    double mass indexed 1 notnull 1 default 0.0
    double charge indexed 1 not null 1 default 0.0
  }
  # …
}
```

A Speed Table looks much like any database table or structure

Example

```
CExtension particles 1.0 {
  CTable quark {
    key id
    double mass indexed 1 notnull 1 default 0.0
    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
  }
  # …
}

package require Particles
quark create t
t index create color
t index create flavor
t set q00001 charge 0.3333 color red flavor strange
...
```

It creates a C extension for managing structured data

Example

```
package require Particles
quark create t
t index create color
t index create flavor
t set q00001 charge 0.3333 color red flavor strange
...
t get q00001
                                  q00001 0.0 0.3333 red strange
t foreach id “q*” {
      puts “quark $id has color [t get $id color]”
   }
                                     quark q00001 has color red
```

Speed Tables can be used as a fast array of structured data.

Compact

```
struct ctable_HashEntry {
    ctable_HashEntry *nextPtr;
    char             *key;
    unsigned int      hash;
};

struct quark: ctable_BaseRow {
    ctable_HashEntry      hashEntry;
    double                charge;
    char                 *color;
    int                   _colorLength;
    int                   _colorAllocatedLength;
    char                 *flavor;
    int                   _flavorLength;
    int                   _flavorAllocatedLength;
};
```

Overhead: one HashEntry per row, two integers per varstring

What’s new in Speed Tables?

Problems in 2006

- A couple of small problems
  - Not much standard library use, lots of ad-hoc structures
  - Assumed 32-bit memory
- And some bigger ones
  - Limited access methods, just a structured array
  - No shared access

Fixing these problems

- Rewritten to use Boost library and made 64-bit clean.
- Secondary indexes and extended search
- Remote speed tables
- Shared memory speed tables.

Searching

- Original search operation simply walked the entire hash table and matched rows
  - Still pretty fast!
  - Unless you want to search on something other than the key.

- Added indexed search (search+) based on skiplists
  - Skip Lists are easy to implement - no rebalancing
  - Skip Lists *potentially* support lockless shared memory access

- William Pugh, 1989
  - ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf

Slightly bigger rows

```
struct ctable_LinkedListNode {
    struct ctable_BaseRow *next;
    struct ctable_BaseRow **prev;
    struct ctable_BaseRow **head;
};

struct quark: ctable_BaseRow {
    ctable_HashEntry      hashEntry;
    ctable_LinkedListNode _ll_nodes[QUARK_NLINKED_LISTS];
    double                charge;
    char                 *color;
    int                   _colorLength;
    int                   _colorAllocatedLength;
    char                 *flavor;
    int                   _flavorLength;
    int                   _flavorAllocatedLength;
};
```

Added: one Linked List Node per index (if used)

Searching

- Search query language very simple and lisp-like
  - {{= fieldname value} {null fieldname} ...}
- Initially, first field in the query was only field that could use an index
- Required user to understand search costs
  - Tedious tweaking
  - Error-prone, especially for automated queries

Query optimizer

- First implemented in Tcl
  - table search -compare [optimize $table {= field value} {< field value} ...]
- Re-implemented in C and vastly improved
  - Score based
  - Modified search based on optimizer
  - Shortcuts like avoiding sorting phase
- Much more convenient and reliable

Filtering

- Compare operation is limited to “AND”, no expressions

- A more complex query language has problems
  - Potentially slow down searches
  - New and fertile source of bugs
  - Lot of work to implement!

C Filters

```
CExtension Filtertest 1.0 {
  CTable airfield {
    key id
    varstring name
    varstring type indexed 1 default GA
    double latitude notnull 1 default 0.0
    double longitude notnull 1 default 0.0
    double altitude notnull 1 default 0.0

    cfilter closer args {double lat double long double range} code {
      double dlat = lat - row->latitude;
      double dlong = long - row->longitude;
      if( ((dlat * dlat) + (dlong * dlong)) <= (range * range) )
        return TCL_OK;
      return TCL_CONTINUE;
    }
}
```

C Filters

```
cfilter closer args {double lat double long double range} code {
  double dlat = lat - row->latitude;
  double dlong = long - row->longitude;
  if( ((dlat * dlat) + (dlong * dlong)) <= (range * range) )
      return TCL_OK;
  return TCL_CONTINUE;
}
```

C Filters

```
airports search \
    -compare { {!= type military} } \
    -filter {closer {*}$mypos 150.0} \
    -array row -code {
        lappend nearby_airfields $row(name)
    }
```

```
int track_filter_closer (Tcl_Interp *interp, struct ctableTable *ctable,
                                          void *vRow, Tcl_Obj *filter, int sequence)
{
    struct track *row = (struct track*)vRow;
    static int lastSequence = 0;
    static double lat = 0.0;
    static double long = 0.0;
    static double 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(Tcl_GetDoubleFromObj (interp, filterList[0], &lat) != TCL_OK)
            return TCL_ERROR;
        if(Tcl_GetDoubleFromObj (interp, filterList[1], &long) != TCL_OK)
            return TCL_ERROR;
        if(Tcl_GetDoubleFromObj (interp, filterList[2], &range) != TCL_OK)
            return TCL_ERROR;
    }

    double dlat = lat - row->latitude;
    double dlong = long - row->longitude;
    if( ((dlat * dlat) + (dlong * dlong)) <= (range * range) )
        return TCL_OK;
    return TCL_CONTINUE;
}
```

Fast data I/O

- read_tabsep
- write_tabsep
- import_postgres_result
- import_cassandra_future

Shared Memory Speed Tables

- (Most of) Speed Table in shared memory
  - Except hash table, management metadata
- Only one process can write
- All other processes are read-only
  - Can perform searches via skiplists
  - Locklessly!

Writer process

- Creates the speedtable
  - speedtable create table master {file ... size ...}
- Hands out tokens to reader processes
  - speedtable attach $pid ==> $list
- Need to have a way to get pids and pass token lists back to reader
  - This is handled outside the speedtable code
- All modifications to speedtable by writer

Reader process

- Requests access to speedtable
  - speedtable create table reader $list
- Performs searches only
- Some search operations not possible
  - E.g. -delete

Lockless

- Reading, adding rows, and updating rows require no locking because of the way skip lists work.
- Deleted rows must be retained until no reader is accessing them
- The master allocates a single word (the cycle) in shared memory, and also assigns a cycle to each reader
- Every time the master deletes a row or rows from the table, it increments the cycle, and stashes the deleted row and the current value for later use

Lockless

- Each time the reader performs a search, it also copies the current value of the cycle to its copy
- Periodically the master “collects” the deleted rows
  - It searches through the readers for the oldest “active” cycle
  - It knows that any rows older than the oldest cycle are not being used by any reader and can be really deleted.
- This work is most of the overhead for the master

Remote Speed Tables

- Client-server protocol
  - Speed Table Transfer Protocol (STTP)
- Connect to remote Speed Table via a socket
  - Works on same machine or over network
- Queries and responses passed over socket as lists
- Except callbacks (search -code, etc…) run locally
  - Bulk data transferred as TSV

Speed Tables API

- Simple syntax
  - ::stapi::connect sttp://localhost:1616/

- Works very well with Shared memory speedtable!
  - ::stapi::connect shared://localhost:1616/
  - Simply makes a remote call to "attach"
  - Redirects everything but "search" to remote master

Speed Tables API

- Very generalizable
- Implemented wrappers around Postgres and Cassandra
- ::stapi::connect sql:///table_name
- ::stapi::connect cass:///keyspace.table

PostgreSQL

- Connect to table

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

- Perform search

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

- Generates and executes SQL

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

Why use STAPI to access PostgreSQL

- Speed Tables very fast, but volatile
- PostgreSQL not volatile, but kind of slow
- Same code can access data multiple ways!
  - Internal Speed Tables loaded from SQL
  - Shared Speed Tables in "cache" process
  - Remote Speed Tables on "cache" host
  - Actual SQL database

Cassandra

- Connect to table

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

- Perform search

  ```
  search -compare {{= isbn 1-56592-00001}} -key k -array row {
      parray row
  }
  ```

- Generates and executes CQL

  ```
  SELECT * FROM stapi_test WHERE isbn = '1-56592-00001';
  ```

- CQL is much more restricted than SQL, so queries are more limited. Some are “hoisted” to fragments of Tcl.

Questions?