Lessons learned from Popeye and Roadrunner

Peter da Silva

Tcl 2019

October 14, 2019

Summary

Popeye is the real-time database for "recent" flights at Flightaware. It uses sqlite for local data storage and exposes a simple TCP command-line API using Tcl lists as the query format.

Over the past year we have brought popeye into production and updated it to wring the last bit of performance we possibly can out of it. A new project for tracking vehicles and planes on the surface, dubbed roadrunner, is being built taking advantage of the lessons learned from popeye.

Overview of popeye (and roadrunner):

Popeye reads events from a data source called "controlstream", which consists of lines organized in key-value format. Each event is used to update a table containing the current status of a flight, a table of positions, and a table of other flight events. It also accepts queries from webservers and other users of the flight data, to provide flight status and flight track information.

```
_hc     1569888000      _hs     2       _c      1569887999      _s      15
type    etms_update     ident   JBU1323 childID 166     clock   1569887991
combid  1569887999-1879 computed_eta    1569898401      dest    KLAX    edt
1569877912      eta     1569899460      facility        etaprediction   fdt
1569874740      id      JBU1323-1569645945-airline-0150:0       msgType
etaprediction   orig    KJFK    otherPedigrees  {aireon wide}   pedigree
wide    predicted_on    1569898401      predicted_on_source     ML-derived EON
recvd   1569887991      reg     N985JT  source  gbt_etas        status  A

_hc     1569974399      _hs     3490    _c      1569974399      _s      3
type    position        ident   YEL5    childID 209     _t      trackInformation
airground       A       alt     40      alt_ft  4000    bitmask 770     clock
1569974393      combid  1569974399-1813 dest    KMGM    facility        KZTL
flightlevel     40      fp      YEL5-1569965569-3-0-209:0       gs      224
gufi    KJ81088300      heading 289     historical      0       lat     32.21583
lon     -86.37361       msgType TFMS    orig    KDAB    otherPedigrees  {aireon
wide}   pedigree        wide    preferred       0       provenance      wide
recvd   1569974397      reg     N585PC  simpleAltitude  40      updateType
Z
```

As can be seen from this example, the number of fields in each record can vary wildly.

Roadrunner will provide the same service for aircraft and other vehicles on the ground. The "surfacestream" data is much more regular and should be easier to manage:

```
_c      1569887999      _s      667     airport KCLT    alt     725     clock
1569888000      first_clock     1569888000      gs      0       heading 0
id      KCLT-1569888000-asdex-4002      ident   Unknown-4002    lat
35.218680       lon     -80.940500      type    track_start     updateType
X

_c      1569887999      _s      668     airport KCLT    alt     725     clock
1569888000      gs      0       heading 0       id      KCLT-1569888000-
asdex-4002      ident   Unknown-4002    lat     35.218680       lon
-80.940500      radius  100     scombid 1569887999-1275 track   KCLT:4002
type    ground_position updateType      X

_c      1569887999      _s      669     aircrafttype    A319    airport KCLT
alt     725     clock   1569888000      fix     KCLT    gs      15      heading
189     hexid   A9942E  id      KCLT-1569887672-hexid-A9942E    ident   AAL1840
lat     35.218150       lon     -80.949500      radius  100     reg     N716UW
scombid 1569887999-1276 squawk  3172    track   KCLT:900        type
ground_position updateType      X

_c      1569887999      _s      670     airport KCLT    alt     725     clock
1569888000      gs      13      heading 71      id      KCLT-1569886558-
asdex-1714      ident   Unknown-1714    lat     35.210330       lon
-80.935420      radius  100     scombid 1569887999-1277 track   KCLT:1714
type    ground_position updateType      X
```

Bottlenecks

One unexpected bottleneck for Popeye is text copying. We expected database updates to take a lot of time, but just generating the SQL commands to populate the database and update the table of current flights had a significant impact on performance.

Since the input is highly irregular, and we only wanted to modify the columns actually present in each record, the alternatives were either execute multiple update statements for each record or generate custom SQL for each row. The former option turned out to be prohibitively slow even for SQLite, so we opted for the latter. Doing it in Tcl involved a lot of list operations, and list/string shimmering, which led to it spending most of its time copying text. Even in C++, using string-views rather than strings, the biggest part of the input process was creating SQL statements for updating the current tracks table. More text copying.

To avoid this string processing, we only generate each possible statement once, create a prepared statement from it, and store it in an N-way tree of TargetNodes.

```
struct TargetNode {
        TargetNode *children[T_NCOLUMNS] = {nullptr};
        sqlite3_stmt *statement[T_NCOLUMNS] = {nullptr};
};
```

This tree can be walked for subsequent records. A new statement is only generated when it needs to extend the tree. This leads to a final set of about 800-1000 prepared statements that are generated out of the 2^60 possible combinations of fields. This is a substantial savings.

Roadrunner reads a much more regular stream, but there are still over 100 unique combinations of fields that need to be handled, so it kept the same code as Popeye to handle this problem.

On the other end, writing to the database has been the primary bottleneck. Even SSD is not fast enough, so we have to keep the Popeye database in RAM disk (tmpfs). Due to the size of the database and how rapidly it's changing periodic snapshots quickly fall behind realtime. Instead, we have over a dozen Popeye nodes and when a new one is spun up or one needs to be rebuilt we temporarily shut down one of the existing nodes and copy the database files over. So the production nodes (in two datacenters) serve as the persistent data for each other.

The single-threaded nature of sqlite is another problem. Even using multiple threads, sqlite only allows access through a connection to one thread at a time, and using multiple connections you're still effectively limited to a single writing thread at a time because the database is locked as a whole. However, we have a workaround.

Roadrunner uses multiple stream readers, each of which handling a subset of the threads. Each reader writes to a shard database that is mounted on the main database using an "attach" sqlite command.

```
ATTACH DATABASE rrdb_$shard.sqlite AS shard$shard;
```

Popeye will be going through the same evolution once we have experience with it in Roadrunner. For roadrunner, since each track is entirely within the bounds of one airport, we shard the database on a hash of the airport, using a simple hash that can be reliably implemented in both C++ and Tcl called the Fowler/Noll/Vo hash[1]:

```
        variable FNV_32_PRIME   [expr 0x01000193]
        variable FNV_32_START   [expr 0x811c9dc5]
        proc bucket {val size} {
                variable FNV_32_PRIME
                variable FNV_32_START

                set result $FNV_32_START

                foreach c [split $val ""] {
                        scan $c "%c" n
                        set result [expr {(($result * $FNV_32_PRIME) & 0xFFFFFFFF)
^ $n}]
                }

                return [expr {$result % $size}]
        }
```

Or in C++:

```
        #define FNV_32_PRIME    ((uint32_t) 0x01000193)
        #define FNV_32_START    ((uint32_t) 0x811c9dc5)
        uint32_t DB::bucket(std::string_view val, int size)
        {
                uint32_t result = FNV_32_START;

                for(auto it = val.cbegin(); it != val.cend(); ++it) {
                        result = (result * FNV_32_PRIME) ^ *it;
                }

                return result % size;
        }
```

Each reader process only attaches the shard it's writing to, and the worker processes that handle user commands, and the background processes that purge old tracks and save completed tracks to a long term PostgreSQL store mount all the shards.

Originally the plan was to completely hide the sharded database structure by using a series of temporary views to emulate flat tables for tracks and events. Unfortunately I have not been able to make the views efficient enough. Accessing the database through a UNION ALL view is at best about three times slower for the typical query, compared to a flat table. Running multiple queries on the table

---

[1] https://en.wikipedia.org/wiki/Fowler-Noll-Vo_hash_function

shards in Tcl and merging the results in the Tcl code is actually a little faster than the original flat tables.

If the shard can be determined by examining the keys (for example, the webserver is pulling up all tracks in a single airport) then only one shard of the table needs to even be examined.

Housekeeping

Cleaning old tracks from the database and archiving completed tracks to PostgreSQL was a bottleneck in the single-threaded popeye, even using tables in a mounted `:memory:` database to hold intermediate results. At times this process stalled the main thread for 8 or 9 seconds during the housekeeping pass. For Roadrunner, these processes are handled by separate processes that have read-only access to the database, and which send lists of flights to purge or mark as archived to the readers over a socket. Since it's using Tcl `fileevents` to track `surfacestream` it can handle requests from other components of Roadrunner the same way. Actually purging tracks that have been (because they've already been archived and are over a day old) now happens in a separate process and finally deleting them typically takes well under 500ms, and even if it takes a few seconds in the housekeeping process that's not a bottleneck.

This means that even without splitting the readers, using separate threads for housekeeping is itself a performance advantage.

Other issues

Popeye is emulating Birdseye which was emulating Trackstream, which used a completely custom set of data structures. The result is that the query formats are a little quirky in places... some search commands use `"-field value"` or `"-range value1 value2"`, others use a list of `{operator field value}` tuples. Similarly, the results in some cases are a simple Tcl list where the caller is expected to know the meaning of each element, others are lists of key-value pairs.

Roadrunner commands will all use the same parser, which will hopefully permit more code re-use in the SQL code generator, and the results will all be key-value lists. For more complex queries, or ones where the web developer is still working on their design, Roadrunner will accept raw SQL, such as: `"select * from target where ident = 'UAL4';"`. This is not intended for long term use since it exposes internal details of the database structure and we need to do ad-hoc rewrites of queries to hide the sharding from customers.