Popeye and Roadrunner

Lessons Learned

Time marches on

- The number of flight events received by FlightAware has increased 40% this year.
- It’s getting harder to keep Popeye up to date.
    - It’s still running faster than real-time, but its lead is shrinking.
- We’re adding a whole new data stream for planes moving on the surface.

Popeye

- Provides real-time view of the skies for webservers and other applications at Flightaware.
- Reads a stream of flight events and maintains a database of tracks and positions.

Surface movement

- Planes move in and out of well defined zones on the ground.
  - Runways, gates, hangers, service areas…
- We can use this to provide information to pilots and ground operators.
- So we want a service to provide easy access to this data.

Roadrunner

- Provides real-time view of the ground for webservers and other applications at Flightaware.

Roadrunner

- Surface movement is being presented as a new event stream
- It’s a much simpler problem than flight events.
  - Tracks are short.
  - Tracks stay in a single airport.
  - Tracks are always well terminated.*
- So Roadrunner solves a simpler problem

* This is a lie, but we make it true.

What this means

- Roadrunner is a good testbed for the next stage of Popeye.

Lessons from Popeye

- A single stream reader and database is hard: SQLite really really likes having a single writer and shared readers.
  - Can’t do parallel writes to the database.
- Popeye has to do all bookkeeping with a single connection.
  - Purging tracks and positions older than a day.
  - Archiving completed tracks to long term database (PostgreSQL).

Applied to Roadrunner

- Roadrunner's database is sharded.
- Each event is hashed by airport ID, and written to a shard of the database.
- Each shard is mounted into the main database when it's opened.

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

- Each stream reader process only writes to its own shard.
- Separate processes handle cleaning and for archiving.

Overview of roadrunner

- Manager process works like ”init”, monitors running processes, restarts them as needed.
  - For example worker processes exit after 1000 user requests.
- Worker process handles user requests.
- Reader process connects to surfacestream and writes its assigned database shard.
- Archiver process writes completed tracks to PostgreSQL
- Bookkeeper process cleans old tracks 24 hours after they complete.

Fowler/Voll/No Hash

- The FNV hash is simple and fast and can be easily implemented in C++ and Tcl.

- It’s not a cryptographic hash. There are potential attacks that we don’t care about, because we’re folding it down to a small integer anyway.

```
#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;
}
```

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

Fowler/Voll/No Hash

- What's that in TCL?

```
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}]
}
```

- Could make a C++ Tcl extension but this is only run once per query when we're restricting the query to one shard, on the airport ID which is a 4-character string, so really it's already overkill.

Cleaner and Archiver

- Querying the database for completed tracks to archive, old archived tracks tracks to purge, and other lint… can take multiple seconds on Popeye. So this is handled by separate processes in Roadrunner.

- Cleaner and archiver have to write to the database, but we don’t want to implement them in the readers. So they bundle up changes and send them to the appropriate stream reader.

- Stream reader process runs the SQL and sends the result back as a Tcl list, similar to the Popeye Trackstream interface. This typically takes tens to hundreds of milliseconds.

- It’s a super simple network database server, just good enough to avoid blocking or delaying the stream readers.

Trackstream

- Popeye provides a simple query language
- Queries are Tcl lists:

```
search -inAir both -originOrDestination KTPA
```

- Responses are Tcl lists:

```
{ident SWA104 prefix {} type B737 suffix {} origin KDCA destination KTPA
departureTime 1539351420 faFlightID SWA104-1539149199-airline-0115 blocked 0
timeout ok timestamp 1539351989 firstPositionTime 1539351463 lowLatitude
38.57555 lowLongitude -77.46361 highLatitude 38.98889 highLongitude -77.04103
longitude -77.46361 latitude 38.57555 groundspeed 381 altitude 198 altitudeFeet
19800 altitudeStatus - updateType TZ altitudeChange D heading 206 arrivalTime 0
estimatedArrivalTime 1539358800} {…} …
```

Surfacestream

- Roadrunner provides a similar interface
- Queries are usually Tcl lists:

```
info KLAX-1572888620-asdex-3366
track KLAX-1572888620-asdex-3366
```

- Will also allow SQL select commands:

```
select lat,lon from target where airport = ‘KCLT';
```

- Responses are Tcl lists:

```
{lon -80.9305 lat 35.22039} {lon -80.93781 lat 35.2141} {lon -80.93192 lat 35.2168}
{lon -80.94842 lat 35.22168} {lon -80.9305 lat 35.22039} {lon -80.94842 lat 35.22168}
{lon -80.94174 lat 35.21776} {lon -80.93779 lat 35.24467} {lon -80.9305 lat 35.22038}
{lon -80.94848 lat 35.22085}
```

Porting back to popeye

- We are currently working on a new stream reader for popeye.
- Due to the higher volume of controlstream the readers will all be spawned from one reader process that reads the stream and hands it over to database writers.
- Because of the convenience of adding a BOOST webserver in C++, the cleaner and archiver will talk to the writers using a REST like interface with JSON responses instead of a TCP servers that reads and writes Tcl lists.

Reading the database

- Changing the schema means changes in the reader-side code.
- First approach was a view that emulated the old schema.

```
CREATE TEMPORARY VIEW inflight (fp, …) AS
   SELECT fp, … FROM shard0.inflight
UNION ALL
   SELECT fp, … FROM shard1.inflight
…;
```

- This was significantly slower, by a factor of 3-20 times depending on the query and schema.

Reading the database

- Roadrunner explicitly runs multiple queries on each request and merges the results in Tcl.
    - SQL queries are minimally edited (target -> shardN.target).
- This was actually 10-15% faster than queries against a single-sharded database.

Performance.

- Writing multiple shards produces basically linear speed up based on the number of shards.
- Reading multiple shards independently and merging the results in Tcl produces a moderate speed up.
- We haven’t re-done the query side of Popeye yet but anticipate similar results.

Unexpected bottleneck

- Text copying is a huge overhead. Even in generating SQL statements to update the inflight table.
- Controlstream was highly irregular, so we couldn’t use a small set of prepared statements.
- We built a tree of observed combinations of keys as we went, and generated and prepared the statement once.
- Ended up with several hundred unique statements after weeks of running.

The big lesson

- Parallelism with SQLITE is tricky but worthwhile.