## Sqlbird

FlightAware is a pretty big Postgres shop. We have a Postgres database server that hosts hundreds of tables with row counts from one row to hundreds of millions, and we run about ten read-only replicas that queries that don't do any writes can be load-balanced to.

These machines are pretty good-sized, by our standards, anyway. 28 64-bit Xeon CPUs, 512 gigs of RAM, high-performance SSDs.

But all SQL that can cause writes must be executed on the master. We also vet queries before allowing them to be load-balanced to the replicas to make sure that latency in bringing the replicas into sync doesn't skew query results and/or cause misbehavior for users.

So we pay close attention to the master's load and performance, because it is a potential performance bottleneck that can't be substantially improved through greater multi-machine parallelization. (Yes there are multi-master SQL database but so far, they are slow.)

In response to performance concerns and load on the primary, in 2012 we began developing a caching system based on speedtables that we called superbird.

Eventually, upon startup Superbird could allocate a bunch of shared memory, replicate tables from Postgres and make it available to clients that could attach the shared memory and perform queries against it.

Tcl code did most of the work, but the importing of Postgres results into a speedtable was done directly in C, so Tcl did not have to "look at" each row of data returned.

Here's an example of what a superbird query looked like.

Queries had to be written in the speedtables query language, but one cool thing was that if a table wasn't replicated on a machine and a speedtables-style query was performed against it, the query would be translated into SQL and performed using the Postgres database behind the program's back, producing identical behavior in handling the results (arrays got filled, break, continue and return inside of loop bodies worked properly, etc.).

This worked pretty well and we made heavy use of it, but it had several drawbacks:
- it loaded all the replicated tables directly into memory
- A single process handled all the replicating of data from Postgres, and with the tables we were replicating, some fairly large, it would take on the order of twenty minutes to start up.
- Likewise when shutting down the program, the operating system would pointlessly flush the shared memory to disk, hammering the machine for no good reason (since we deleted the shared memory backing storage file anyway), and it could take ten minutes or more to shut down.
- Speedtables, though very fast, has a limited and unique-to-itself query language that, among other limitations, doesn't support joins.

We did over time bring Superbird into production and to a pretty high level of reliability. It is robust at reconnecting to the database server, retrying many times if necessary, should it lose its connection, for instance. (With it taking 30 minutes to restart, it's particularly important that the program be reliable even when there may be intermittent problems with the database servers, for instance.)

So we considered it to be a win and an important tool in our arsenal.

But we weren't completely satisfied for all the aforementioned reasons.

In mid-2017 we started looking at whether it might be possible to do the same sort of thing, but to use SQLite for the cache instead of speedtables.

Since SQLite was storage-based, a restart of the replicator wouldn't require everything to be re-replicated from scratch.

SQLite supports several different journalling modes, ways of providing support for transactions. Write-ahead log, or WAL mode
- is significantly faster than journal mode
- it allows unlimited concurrent readers with a single concurrent writer.
- It does many fewer fsyncs

That seemed like it would work for us, and the single writer limitation isn't a problem at all for this purpose.

Since SQLite is SQL, there was the distant possibility of being able to run the same SQL queries in Postgres and SQLite without any changes.

Creating SQLbird

The first step was to read table and index definitions, etc, out of Postgres.

The SQL standard defines something called the "information schema", which is a relatively portable way to obtain table definitions and the like from a SQL database.

The Postgres system catalogs, on the other hand, are unique to Postgres.

However the information schema doesn't contain information about Postgres-specific features. Also we already had code for digging table and index and view definitions, and the like, out of Postgres.

And there's an easy way to find out what queries to perform on the Postgres system tables to get the definitions. The interactive "psql" tool will connect to a Postgres database and provides ways to execute SQL queries and has various interactive capabilities to obtain and display the definition of a table, a view, a table's indexes, etc. Fire up psql with the "—echo-hidden" option and it will emit all of the queries it makes against system tables to, for instance, obtain the definition of a table.

Because we were already reading table definitions out of Postgres for Superbird by querying the Postgres system tables, we continued doing it this way.

OK so without much work we were obtaining the table definitions. Furthermore, we understood them. We knew the columns and data types and the columns being indexed, etc; we didn't just have the table definitions themselves.

Next we started to examine support for data types in SQLite and PostgreSQL.

At first it looked a little scary. Postgres has a rich set of data types, including user-defined ones, while SQLite has only four: integer, real, text and blob. Could the postgres types, or at least the ones we needed, be translated and meaningfully used with SQLite's limited types?

The answer was yes. Postgres's many integer data types could be mapped to SQLite's single integer one (it is stored variable-width from 8 to 64 bits, so it can store big integers), real and double precision can be mapped to SQLite's "real", and booleans to integers (more on this later).

Almost all the other types… varchar, timestamps, intervals, dates, internet addresses, etc, could be mapped to text.

Soon we had code reading the SQL definitions from Postgres, creating new ones for SQLite and were running them. Also we could create the indexes. A small number of Postgres indexes were too fancy for SQLite and would error. In that case we report the problem, skip that index, and move on. Although you would want to look at these keenly to make sure you aren't taking a severe performance penalty somewhere from a missing index, we also provide a way to create additional indexes in SQLite that aren't there in postgres.

The idea was that we would initially select all the rows from the tables initially then, for big tables at least, use a trigger-maintained "changed" column and fetch only updates for efficiency's sake.

SLIDE

We had something working fairly quickly. You could make a list of tables you wanted replicated and how frequently they should be refreshed. You could telnet into the console port and manually refresh tables, start replicating new tables, drop tables, and so forth. It seemed pretty cool.

But there was still no concurrency during startup, and some tables needed to accessed with different credentials from others, and the whole list-of-tables thing was too flat. Different machines would need to replicate different tables and whatnot; two different applications might need different tables replicated, and having them all defined flat and in one place, you could see how it was going to cause problems.

We were able to solve both problems by creating replication sets. A higher level of specification that would name one or more tables into a replication set. Each replication set would be stored in its own SQLite database file and would have its own program to replicate the data from the database to it.

SLIDE

Since each database file had its own replicator, called babybird, a manager daemon, sqlbirdd, could start babybird for each replication set and they could proceed concurrently. This cut startup time to the time required for the longest single table to replicate and get its indexes created.

When attaching multiple SQLite databases as different schema, SQLite will find tables in whatever schema they're in, without requiring the schema to be explicitly referenced, which is ideal as long as there are no instances of the same table name in multiple schema. With this approach there is no cognitive penalty on developers to explicitly call out the repset schema names in in their queries, or the likelihood of breaking code if a table is moved from one replication set to another for operational reasons. Three cheers for SQLite!

sqlbirdd runs the babybirds and collects and logs their output. It looks for a particular message from each babybird saying that it has initially caught up, that it has successfully replicated or refreshed everything that is there. Only after all babybirds have caught up does sqlbirdd background itself. This is good startup behavior so that, for instance, sqlbird can be fully up before allowing the system startup process to startup the applications that require sqlbird to be there, for instance the webserver.

Getting sqlite modified

Although we were able to map booleans to integers, a fair bit of our SQL failed on SQLite because of a lack of the true and false keywords in SQLite's implementation of SQL. (IS TRUE, IS FALSE, IS NOT TRUE, IS NOT FALSE).

I reached out to Richard Hipp, the primary author of SQLite, and explained the issue, and he graciously added support for this, and very quickly as well. It's good to have friends in high places.

He had also previously added support for -withoutnulls, something we created and use a fair bit in Pgtcl, where Tcl array variables being created from query results, instead of being set to an empty string when null, the elements are instead unset, and hence can be tested for existence with info exists distinctly from being an empty string or whatever, thus providing Tcl programs with true out-of-band null detection.

Modifying Pgtcl

libpq is the C interface to Postgres. Normally when you do a select, all the results are buffered in memory locally by libpq before being made available to the caller. This is normally probably good, but when you are getting back millions of rows it does use up a lot of memory and the program can't be doing any incremental work while the results are streaming back.

The Postgres team added a row-by-row option, where you get incremental returns of results, to libpq and Peter da Silva extended the Tcl interface to Postgres, Pgtcl, to support it. We use this so babybird can get results back row-by-row and do work on the SQLite database while rows are coming over the wire from the Postgres server.

One thing we really liked about the SQLite Tcl interface is how you can reference variables with the :var notation. I had avoided supporting this in Pgtcl because it would require tokenizing the SQL and I felt it was too much overhead. But after using SQLite a good bit, I really liked the feature and thought Pgtcl should have it.

So Peter added the -variables option to Pgtcl's pg_select and friends, which provides the same variable-referencing behavior that SQLite has.

You might say we're pulling SQLite and Postges closer together.

So we started using sqlbird in various places in our infrastructure and it was working out pretty well.

But there was this sort of tantalizing idea that maybe, as with superbird, it might be possible to perform a select against sqlite but fall back and run the identical query on postgres if for some reason it didn't work on sqlite (like, in particular, some needed table not being there.)

With the changes to SQLite and to Pgtcl, the vasty majority of our SQL queries, including large and complicated ones, that work in Postgres will run unmodified in SQLite. And we succeeded. Sqlbird's apex user function, sqlbird::select, will try the SQL query first in SQLite and fall back to PostgreSQL if it doesn't work.

SLIDE

To pull this off, sqlbird::select has to be very careful and use uplevel and upvar and catch just exactly right, but the result is that a Tcl code body can be iterated over sqlite results or postgres results, transparently, with correct behavior in the code body for continue, break and return.

This is something you can do with Tcl that you can't do in almost any other language. You can't do it in Java or Scala or Swift or Python or Perl. And that's pretty cool…

The current status is that it works and it's running on at least 60 of our servers. We have replaced superbird with sqlbird successfully in some large subsystems and are in the process of the biggie, replacing all of superbird with sqlbird on the webservers. The work has been done and it's in test and pretty soon it'll be in production. Jon Cone has done much of this work.

Jon also extended sqlbird to support replicating views between postgres and sqlite.

We are willing to release all this as open source. We think it would be useful to a lot of people and be something where even a non-Tcl shop would install Tcl just to get these capabilities. The only problem is that it's got some flightaware-isms that need to be removed, particularly around some configuration-loading stuff and monitoring stuff. But in a way where it would be pluggable or something, so there aren't two codebases to maintain.

The sqlbird command line tool finds all of the sqlbird sqlite databases and fires up the sqlite interactive program with the right command line arguments to attach them all. It's handy.

Peter has also written pg_deltaflood, a program that follows the Postgres WAL log and can update SQLite from that. We haven't plugged it all together yet and really tried to use it, but it holds the promise of keeping the SQLite replicas much more up to date and with less overhead on the postgres server than our polling approach.

DEMO

QUESTIONS

The sqlbird-replicated database files are mounted read-only by default and the tables should be considered read-only. Updates to these tables need to be performed directly to the Postgres server and will eventually be reflected in the sqlbird replicas when the scheduled updates are performed.

A word about time… you might have noticed that we converted date types to varchar rather than trying to clock scan them to long integers or whatever. This seems to be the sqlite approach. Date/times in SQL can be compared lexically and work properly for equality, greater than and less than, but intervals cannot easily be determined.

we are, I believe, in general, over-indexed insofar as many of the indexes on the tables serve to speed up queries that sqlbird isn't being asked to do, anyway. An audit to find columns that are being indexed that don't improve performance would improve startup times and reduce the overhead of maintaining the tables. The largest tables tend to have a lot of indexes, so this could be a big win. The downsides of course are the time required for the audit and the risk of removing an index that turns out to be important under some unforeseen circumstance.

```sql
CREATE OR REPLACE FUNCTION onupdate_changed() RETURNS trigger AS $$
  BEGIN
    NEW.changed := (current_timestamp at time zone 'UTC');
    RETURN NEW;
  END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER mytable_update BEFORE UPDATE ON mytable FOR EACH ROW EXECUTE
PROCEDURE onupdate_changed();
```

### accessing the sqlbird SQLite cache databases from your tcl program

```tcl
package require sqlbird

sqlbird_open

sqlbird eval "select * from ..." row {}
```

This is quite analagous to using pg_select.

`

Updates by application programs to the SQLite database and writing them back to the PostgreSQL database, etc, are not supported. Make updates by updating the PostgreSQL tables and letting sqlbird replicate the changes back by its usual mechanism. (Feel free to create your own local SQLite databases that you do manage and update from your application, though... multicom does this, for example.)

All PostgreSQL requests are automatically load-balanced on the presumption that if you are willing to get it from sqlbird, where it may be behind a few seconds or more, you're ok with getting it from one of the database replicas, which also tend to run a few seconds behind.

If -withoutnulls is present, null values aren't set into the array as empty strings.

SQL queries should be done using the SQLite variable substitution semantics aka the pg_select _-variables_ semantics. So an example might be

```tcl
        sqlbird::select -withoutnulls "select * from mytable where id = :user_id" row {
                parray row
        }
```

```tcl
sqlbird::define_repset web \
        -schema web \
        -consoleport 1282 \
        -dbfile /var/db/sqlbird/web.db
        -tables {
                10s {flightaware_config users user_roles user_permissions}
                5m {airlines metars tafs}
                4h {aircraft idents faa_master all_airports aircraft_registry_joined}
        }
```

Recognizing that for running sqlbirdd in a docker container you don't want it to background itself, sqlbirdd checks to see if it is in a docker environment. If so then it forces the -foreground option.