Streaming replication between database engines via Tcl

Peter da Silva

Tcl 2017

October 18, 2017

Summary

FlightAware currently uses speedtables to cache certain frequently accessed tables on webservers and other application servers. A similar utility using sqlite3 as the local cache is in development. Currently this cache is initialized and updated by having each server maintain a connection to the database and re-reading the tables into the speedtables (or sqlite3) cache on a regular basis. For the sqlite3 version, we are going to use PostgreSQL replication facilities to keep the sqlite3 databases in sync with the master.

pg_recvlogical

PostgreSQL has a mechanism to dump database changes in an arbitrary streaming format. When changes are stored in the WAL (write-ahead log) they can also be queued up using an "output plugin" in a slot that can be read by an application. As distributed, PostgreSQL provides an application pg_recvlogical that dumps the stream from a given slot onto its standard output, and a sample output plugin that dumps the changes in a human-readable format, which is useful for debugging the system but not particularly parseable.

deltaflood

http://github.com/flightaware/pg-deltaflood

Deltaflood runs as an output plugin in the PostgreSQL server. It is called as each change is added to the write-ahead log and passed a tagged C structure describing the change. It selects the events that are of interest and generates a stream of change records as key-value pairs. This is queued up in an output slot in the database server, and is extracted by calling pg_recvlogical and reading the records it dumps to standard output.

Each generated record is a tab-separated line in the following format:

```
_table  zzz     _xid    43559077 _action  update   a       fox69   b       hen39
_table  zzz     _xid    43559521 _action  update   a       fox15   b       hen43
_table  zzz     _xid    43559964 _action  delete   a       fox63
_table  zzz     _xid    43559964 _action  replace  a       fox39
_table  zzz     _xid    43560388 _action  update   a       fox84   b       hen2
_table  zzz     _xid    43560797 _action  update   a       fox38   b       hen53
_table  zzz     _xid    43561238 _action  insert   a       fox14   b       hen33
```

This is relatively straightforward: table name, transaction ID, and action, followed by the primary key and (for insert or update) non-key columns. The possible actions are:

insert	Insert a row in the database
delete	Delete a row from the database
replace	tag row for replacement
update	Update a row in the database

The replace action is only generated when the primary key for a row will be changed in the immediately following update action, and contains the original primary key.

Newlines, tabs, and a few other characters are encoded with backslash escapes suitable for decoding with `[subst -nocommands -novariables ...]`.

Daystream

Daystream is a library and a data format that is used extensively within FlightAware for transferring data streams such as flight events between applications. Each event is a row of tab-separated fields of alternating key-value pairs, similar to the Tcl "array get" format. The first four fields are a tagged timestamp consisting of a UNIX clock and a sequence number.

```
_c    1504631452  _s    12    key1  val1  key2  val2  ...
```

The library uses local configuration information to locate the requested stream, either local files or a remote server that contains the requested files, and then calls a user-provided callback for each line. The user can start reading at the end of the stream and just get new events, or start at a specific timestamp.

The last record in each file looks like this:

```
_c    1504223999   _s    9999   _t    switch file   /path/filename.tsv
```

Which tells the library where to continue when it's reading from local files.

Publishing the changes via daystream will allow multiple hosts to track database changes by subscribing to "deltastream", without loading the database server down with multiple pg_recvlogical connections.

pg_sqlite

http://github.com/flightaware/Pgtcl

The sqlite extension to the Pgtcl package adds a new command, pg_sqlite, that can be used to rapidly copy data from PostgreSQL to Sqlite3.

```
set res [$pgdb exec "SELECT * FROM TABLENAME;"]
pg_sqlite $sqlitedb import_postgres_result $res \
      -into tablename \
      -as {col type col type ...} \
      -pkey {col col col}
pg_result $res clear
```

The column definition and primary key are required, and can usually be generated by querying the databases if they are not known (eg, in a generic application).

This extension is only incorporated if Sqlite3 is installed on the system (which is usually going to be the case in a typical Tcl installation). It can be disabled if necessary with the usual configure option --without-sqlite3.

deltastream and deltamirror

deltastream runs pg_recvlogical and feeds the output into daystream. It could write directly to SQL but using daystream as an intermediate allows multiple sqlite caches to feed from a single logical replication slot in the PostgreSQL server.

The daystream rows generated are simply the rows received from the output plugin with the standard daystream timestamp prepended.

```
_c 1504640972 _s 6 _table zzz _xid 43691778 _action update a fox13 b hen63
_c 1504641002 _s 7 _table zzz _xid 43692249 _action update a fox15 b hen62
_c 1504641032 _s 12 _table zzz _xid 43692711 _action update a fox86 b hen64
_c 1504641063 _s 0 _table zzz _xid 43693179 _action delete a fox65
_c 1504641093 _s 5 _table zzz _xid 43693653 _action update a fox50 b hen49
```

deltamirror reads a daystream feed and updates an sqlite3 database. It's a very thin wrapper around sqlite3 that implements the replication operations directly as inserts, deletes, and updates. Each transaction also updates a copy of the most recent timestamp in a table in the sqlite3 database, which allows the stream to be restarted where it left off.

Because sqlite3 does not prioritize support for concurrent access, we are currently working on tuning the transaction size (how many deltas are bundled together into a single transaction). It may be necessary to incorporate deltamirror into the event loop in the final application.

Bringing it all together

Initial configuration of the sqlite database is performed by querying the PostgreSQL schema and generating the sqlite schema from that data. Querying the PostgreSQL schema is an expensive operation that, if performed repeatedly, can hold of vacuuming on the schema tables with unfortunate results. So we cache the result of the queries in a table and use that for subsequent lookups.

Once the schema is cloned, we use pg_sqlite to populate the Sqlite3 tables from PostgreSQL, and start up deltamirror to keep them updated.

Tcl is a particularly good platform for this, because not only is it easy to write extensions for, but also the natural data structures in Tcl tend to encourage a clear "best way" of doing things, so the existing extensions have a low semantic distance from each other... there's no messing around converting between arrays and hashes and other collections. This leads to very "straight line" glue code that doesn't spend a lot of time rearranging data to feed into the next step. Thank You EIAS and key-value lists.

One More Thing

Getting the sqlite3 database handle from the database object name requires a little bit of parkour. It's possible because the first element of the userdata field for an sqlite database command is a pointer to the sqlite3 database handle. This appears to be deliberate:

```
struct SqliteDb {
       sqlite3 *db; /* The "real" database structure. MUST BE FIRST */
       // other stuff we don't look at...
};
```

One problem is the user may not have passed pg_sqlite a valid sqlite3 command name, so it needs to verify this before blindly grabbing whatever data is referenced by the userdata pointer. That it's a command is easy enough to check, since we need to look it up to get the userdata field. Making sure that it's an sqlite3 command takes a little more work. Since the ObjProc for all sqlite3 commands should be the same, we create an sqlite3 database connection and cache the command's ObjProc, which can then be compared with the ObjProc for the command passed to pg_sqlite. This is all a bit ad-hoc, but it's safe: if it fails it will fail by refusing to accept valid

sqlite3 handles rather than by blindly following a pointer to random memory and treating it as an sqlite3 database and initiating total protonic reversal.