Simple declarative schema migration for SQLite

By David Rõthlisberger (https://david.rothlis.net) and William Manley (https://blog.williammanley.net).
Published 30 Apr 2022.

At my company (https://stb-tester.com) we use a small SQLite database. We define the schema in a single file with SQL *CREATE TABLE* statements. If we add tables, columns, or indexes, our application will create them automatically the next time it starts up.

This is superior to explicit database migration scripts:

- We (usually) don't need to write the database migration SQL manually.
- This brings some of the benefits of schemaless databases to SQL, particularly around speed of development / ease of experimentation.
- The auto-migration code gives us some guarantees that the resulting database schema will match our SQL file.
- We can downgrade too, by dropping tables or columns (if we don't mind losing data). This is important for CI where switching between branches can cause the schema to change regularly.

We've been using this since 2019 and it works well, though I must admit our database is small (a dozen tables, ~40MB) and it doesn't change *that* often (65 changes to our schema in those 3 years, according to *git log*).

How it works

We define our database schema in a single file —let's call it *schema.sql*— with normal *CREATE TABLE* and *CREATE INDEX* statements.

When our application starts up, our migrator creates a new in-memory database and executes the schema to create a "pristine" or "desired" version of the database:

```
schema = open("schema.sql").read()
pristine = sqlite3.connect(":memory:")
pristine.executescript(schema)
```

We query sqlite’s internal sqlite_schema (https://www.sqlite.org/schematab.html) table to get the tables from both the “pristine” and “actual” databases:

```
pristine_tables = dict(pristine.execute('''\
    SELECT name, sql FROM sqlite_schema
    WHERE type = "table" AND name != "sqlite_sequence"''').fe
tables = dict(db.execute('''\
    SELECT name, sql FROM sqlite_schema
    WHERE type = "table" AND name != "sqlite_sequence"''').fe
```

Then we can work out new or removed tables:

```
new_tables = set(pristine_tables.keys()) - set(tables.keys()
removed_tables = set(tables.keys()) - set(pristine_tables.key
```

The above query gives us the *CREATE TABLE* sql, which we can execute to create the new tables:

```
sqlite> select name, sql from sqlite_schema where type = "tal
name|sql
Node|CREATE TABLE "Node"(...)
...
```

Similarly for indexes:

```
sqlite> select name, sql from sqlite_schema where type = "inc
name|sql
Node_node_id|CREATE UNIQUE INDEX Node_node_id on Node(node_ic
...
```

To detect changes to existing tables we use PRAGMA table_info (https://sqlite.org/pragma.html#pragma_table_info), which returns a list of the table’s columns:

```
sqlite> pragma table_info(Node);
cid|name|type|notnull|dflt_value|pk
0|node_oid|INTEGER|1||1
1|node_id|TEXT|1||0
...
```

If there are new or changed columns, we follow the 12 step procedure (https://www.sqlite.org/lang_altertable.html#otheralter) in the SQLite documentation. In short we create a new table, copy the data from the old table into the new table, drop the old table, and rename the new table. This particular sequence is important to avoid breaking foreign keys.

We use a similar technique (querying *sqlite_schema* and *table_info*) to generate an Entity Relationship Diagram using Graphviz (https://graphviz.org/), for our documentation.

ORM agnostic

With this technique you can define your database schema however you like. We use *CREATE TABLE* statements in a single sql file, but maybe you use an ORM that can create the database tables based on your ORM-ey code. You just need to create a new temporary database from scratch; our technique then compares the current database versus the new temporary database.

Limitations

The schema changes that we can make are limited to the following operations:

1. Adding a new table.
2. Adding, deleting or modifying an index.
3. Adding a column to an existing table as long as the new column can be NULL or has a DEFAULT value specified.
4. Changing a column to remove NULL or DEFAULT as long as all values in the database are not NULL.

5. Changing the type of a column (note that SQLite's typing is, uh, flexible (https://sqlite.org/flextypegood.html)).

And if you don't mind losing data:

6. Dropping a table.
7. Dropping a column.

(For those you need to opt in by specifying *allow_deletions=True*.)

Our migrator doesn't support triggers & views — not for any fundamental reason, it's just that we don't use SQLite triggers & views in our application.

Our migrator may change the values of rowid (https://www.sqlite.org/rowidtable.html) columns (which are generated internally by SQLite) because we call VACUUM (https://www.sqlite.org/lang_vacuum.html) to re-pack the database file.

Our migrator doesn't do *data* migrations (where you're changing the format of existing data or populating new columns based on other columns). You still need to write those manually.

Manual migrations

In our initial design, our migrator would check the *user_version (https://www.sqlite.org/pragma.html#pragma_user_version)* pragma and refuse to auto-migrate if *user_version* had changed. In theory we would bump *user_version* in our *schema.sql* file when the changes were too complex for the migrator. In practice, we never used this; instead we write explicit SQL to bring the database into a state that the auto-migrator will be happy with.

For example, if we have removed a table from the schema and we don't care about the data in it, we write *DROP TABLE IF EXISTS* before running the migrator. This is idempotent so it won't hurt if it is run again. In due course this migration can be removed from the code, after the change has been rolled out to all the relevant servers.

Data migrations can be run after the auto-migrator, once the new tables & columns are in place.

For example:

```
db = sqlite3.connect("mydatabase.sqlite3")
schema_sql = open("schema.sql").read()
v, = connection.execute("PRAGMA user_version").fetchone()
if v == 0:
    # Initialising database from scratch.
    with DBMigrator(db, schema_sql) as migrator:
        migrator.migrate()
elif v == 1:
    with DBMigrator(db, schema_sql) as migrator:

        # Manual migration: Drop obsolete tables so that `mig
        # doesn't complain in prod where `allow_deletions=Fa
        db.execute("DROP TABLE IF EXISTS MyObsoleteTable")

        # Now do the automatic migration:
        migrator.migrate()

        # Additional data migration to populate new "end_time
        # Note the careful WHERE clause to make this idempote
        db.execute("""\
            UPDATE Task SET end_time = :now
            WHERE result IS NOT NULL AND end_time IS NULL"""
            {"now": time.time()})

else:
    raise RuntimeError(
        f"Database is at version {v}. This version of softwa
        "only supports opening versions 0 or 1. Bailing out 
        "to avoid data loss.")
```

This example runs the manual & automatic migrations in the same transaction (in case that's important to you). `DBMigrator` is a Python context manager so that we can run the last step of the 12 step procedure (https://www.sqlite.org/lang_altertable.html#otheralter) (re-enabling foreign key constraints) after all your manual migrations, because that step can only be run after committing the transaction.

Continuous Integration

Our unit tests (for our product, not for this migration system specifically) create an in-memory database from scratch each time, using something like this:

```
@pytest.fixture()
def tmpdb():
    return MyDatabase(sqlite3.connect(":memory:"))
```

Since there's nothing to migrate *from*, these unit tests aren't testing the schema migration.

We also have some integration tests (our product is a distributed IoT-style system so the integration tests include a real "portal" or central server and real "nodes" running on our product's actual hardware). These integration tests re-use a database that persists across CI runs. Our CI might run tests from different branches with divergent schemas, so before CI we run the migrator with *allow_deletions*=*True* to downgrade to the schema on our "main" branch. Then the actual migration will be exercised when our application starts up.

Implementation

Here's the code — I extracted it from our proprietary codebase but I haven't tested it in isolation:

- migrator.py
- Unit tests

See the code in “Manual migrations”, above, for a usage example.

Credits

Design & code by my colleague William Manley (https://blog.williammanley.net), partly inspired by liyanchang’s comment on Hacker News (https://news.ycombinator.com/item?id=19881330).

Related work

Migra (https://github.com/djrobstep/migra) (for Postgres), **Skeema (https://github.com/skeema/skeema)** (for MySQL/MariaDB), and **sqldef (https://github.com/k0kubun/sqldef)** (for MySQL, Postres, Microsoft SQL Server, **and** SQLite!) are all quite similar to what I have described.

Are they ORM agnostic? I think so: they can all connect to an existing database to dump the database’s schema to sql files, so you can create the “desired” schema any way you want. Skeema expects you to commit the dumped sql files to your version control, and editing them is the canonical way of changing the schema, but I’m sure that isn’t strictly necessary.

Skeema has a (paid) GitHub integration that will automatically post a comment on pull requests to show the generated migration, and maybe it warns about unsafe (data losing) migrations? Sounds pretty neat.

At the time we implemented our migrator, sqldef didn’t yet support SQLite. Even now, it isn’t clear (https://github.com/k0kubun/sqldef/blob/v0.11.50/schema/generator.go) that slqdef implements SQLite’s 12 step procedure (https://www.sqlite.org/lang_altertable.html#otheralter) for safely altering tables. Like our migrator, sqldef asks the database for its own schema (*select sql from sqlite_schema where tbl_name = ?*) but to detect added/removed columns sqldef uses its own SQL parser instead of querying *PRAGMA table_info* (this isn’t necessarily a bad thing, but it’s complexity that isn’t needed for our use case).

sqlite-utils (https://sqlite-utils.datasette.io/en/stable/#) (from Simon Willison's Datasette (https://datasette.io/)) creates/modifies tables automatically based on the CSV/JSON data that you're importing, and it does implement SQLite's 12 step procedure (https://sqlite-utils.datasette.io/en/stable/python-api.html#python-api-transform) for altering tables.

Comments

Hacker News comments on this article here (https://news.ycombinator.com/item?id=31249823).