sqlite-utils documentation

Release 3.15

Simon Willison

Aug 09, 2021

Contents

Python utility functions for manipulating SQLite databases

This library and command-line utility helps create SQLite databases from an existing collection of data.

Most of the functionality is available as either a Python API or through the `sqlite-utils` command-line tool.

sqlite-utils is not intended to be a full ORM: the focus is utility helpers to make creating the initial database and populating it with data as productive as possible.

It is designed as a useful complement to Datasette.

CHAPTER 1

Contents

1.1 Installation

`sqlite-utils` is tested on Linux, macOS and Windows.

1.1.1 Using Homebrew

The *sqlite-utils command-line tool* can be installed on macOS using Homebrew:

```
brew install sqlite-utils
```

If you have it installed and want to upgrade to the most recent release, you can run:

```
brew upgrade sqlite-utils
```

Then run `sqlite-utils --version` to confirm the installed version.

1.1.2 Using pip

The sqlite-utils package on PyPI includes both the *sqlite_utils Python library* and the `sqlite-utils` command-line tool. You can install them using `pip` like so:

```
pip install sqlite-utils
```

1.1.3 Using pipx

pipx is a tool for installing Python command-line applications in their own isolated environments. You can use `pipx` to install the `sqlite-utils` command-line tool like this:

```
pipx install sqlite-utils
```

1.2 sqlite-utils command-line tool

The `sqlite-utils` command-line tool can be used to manipulate SQLite databases in a number of different ways.

1.2.1 Running SQL queries

The `sqlite-utils query` command lets you run queries directly against a SQLite database file. This is the default subcommand, so the following two examples work the same way:

```
$ sqlite-utils query dogs.db "select * from dogs"
$ sqlite-utils dogs.db "select * from dogs"
```

Returning JSON

The default format returned for queries is JSON:

```
$ sqlite-utils dogs.db "select * from dogs"
[{"id": 1, "age": 4, "name": "Cleo"},
 {"id": 2, "age": 2, "name": "Pancakes"}]
```

Newline-delimited JSON

Use `--nl` to get back newline-delimited JSON objects:

```
$ sqlite-utils dogs.db "select * from dogs" --nl
{"id": 1, "age": 4, "name": "Cleo"}
{"id": 2, "age": 2, "name": "Pancakes"}
```

JSON arrays

You can use `--arrays` to request arrays instead of objects:

```
$ sqlite-utils dogs.db "select * from dogs" --arrays
[[1, 4, "Cleo"],
 [2, 2, "Pancakes"]]
```

You can also combine `--arrays` and `--nl`:

```
$ sqlite-utils dogs.db "select * from dogs" --arrays --nl
[1, 4, "Cleo"]
[2, 2, "Pancakes"]
```

If you want to pretty-print the output further, you can pipe it through `python -mjson.tool`:

```
$ sqlite-utils dogs.db "select * from dogs" | python -mjson.tool
[
    {
        "id": 1,
        "age": 4,
        "name": "Cleo"
    },
    {
        "id": 2,
        "age": 2,
        "name": "Pancakes"
    }
]
```

Binary data in JSON

Binary strings are not valid JSON, so BLOB columns containing binary data will be returned as a JSON object containing base64 encoded data, that looks like this:

```
$ sqlite-utils dogs.db "select name, content from images" | python -mjson.tool
[
    {
        "name": "transparent.gif",
        "content": {
            "$base64": true,
            "encoded": "R0lGODlhAQABAIAAAAAAAP///yH5BAEAAAAALAAAAAABAAEAAAIBRAA7"
        }
    }
]
```

Nested JSON values

If one of your columns contains JSON, by default it will be returned as an escaped string:

```
$ sqlite-utils dogs.db "select * from dogs" | python -mjson.tool
[
    {
        "id": 1,
        "name": "Cleo",
        "friends": "[{\"name\": \"Pancakes\"}, {\"name\": \"Bailey\"}]"
    }
]
```

You can use the `--json-cols` option to automatically detect these JSON columns and output them as nested JSON data:

```
$ sqlite-utils dogs.db "select * from dogs" --json-cols | python -mjson.tool
[
    {
        "id": 1,
        "name": "Cleo",
        "friends": [
            {
                "name": "Pancakes"
            },
            {
                "name": "Bailey"
            }
        ]
    }
]
```

Returning CSV or TSV

You can use the `--csv` option to return results as CSV:

```
$ sqlite-utils dogs.db "select * from dogs" --csv
id,age,name
```

(continues on next page)

(continued from previous page)

```
1,4,Cleo
2,2,Pancakes
```

This will default to including the column names as a header row. To exclude the headers, use `--no-headers`:

```
$ sqlite-utils dogs.db "select * from dogs" --csv --no-headers
1,4,Cleo
2,2,Pancakes
```

Use `--tsv` instead of `--csv` to get back tab-separated values:

```
$ sqlite-utils dogs.db "select * from dogs" --tsv
id  age     name
1   4       Cleo
2   2       Pancakes
```

Table-formatted output

You can use the `--table` option (or `-t` shortcut) to output query results as a table:

```
$ sqlite-utils dogs.db "select * from dogs" --table
  id    age  name
----  -----  --------
   1      4  Cleo
   2      2  Pancakes
```

You can use the `--fmt` option to specify different table formats, for example `rst` for reStructuredText:

```
$ sqlite-utils dogs.db "select * from dogs" --table --fmt rst
====  =====  ========
  id    age  name
====  =====  ========
   1      4  Cleo
   2      2  Pancakes
====  =====  ========
```

For a full list of table format options, run `sqlite-utils query --help`.

Returning raw data, such as binary content

If your table contains binary data in a `BLOB` you can use the `--raw` option to output specific columns directly to standard out.

For example, to retrieve a binary image from a `BLOB` column and store it in a file you can use the following:

```
$ sqlite-utils photos.db "select contents from photos where id=1" --raw > myphoto.jpg
```

Using named parameters

You can pass named parameters to the query using `-p`:

```
$ sqlite-utils query dogs.db "select :num * :num2" -p num 5 -p num2 6
[{":num * :num2": 30}]
```

These will be correctly quoted and escaped in the SQL query, providing a safe way to combine other values with SQL.

UPDATE, INSERT and DELETE

If you execute an `UPDATE`, `INSERT` or `DELETE` query the command will return the number of affected rows:

```
$ sqlite-utils dogs.db "update dogs set age = 5 where name = 'Cleo'"
[{"rows_affected": 1}]
```

SQLite extensions

You can load SQLite extension modules using the `--load-extension` option, see *Loading SQLite extensions*.

```
$ sqlite-utils dogs.db "select spatialite_version()" --load-extension=spatialite
[{"spatialite_version()": "4.3.0a"}]
```

Attaching additional databases

SQLite supports cross-database SQL queries, which can join data from tables in more than one database file.

You can attach one or more additional databases using the `--attach` option, providing an alias to use for that database and the path to the SQLite file on disk.

This example attaches the `books.db` database under the alias `books` and then runs a query that combines data from that database with the default `dogs.db` database:

```
sqlite-utils dogs.db --attach books books.db \
    'select * from sqlite_master union all select * from books.sqlite_master'
```

1.2.2 Querying data directly using an in-memory database

The `sqlite-utils memory` command works similar to `sqlite-utils query`, but allows you to execute queries against an in-memory database.

You can also pass this command CSV or JSON files which will be loaded into a temporary in-memory table, allowing you to execute SQL against that data without a separate step to first convert it to SQLite.

Without any extra arguments, this command executes SQL against the in-memory database directly:

```
$ sqlite-utils memory 'select sqlite_version()'
[{"sqlite_version()": "3.35.5"}]
```

It takes all of the same output formatting options as *sqlite-utils query*: `--csv` and `--csv` and `--table` and `--nl`:

```
$ sqlite-utils memory 'select sqlite_version()' --csv
sqlite_version()
3.35.5
$ sqlite-utils memory 'select sqlite_version()' --table --fmt grid
+--------------------+
| sqlite_version()   |
+====================+
| 3.35.5             |
+--------------------+
```

Running queries directly against CSV or JSON

If you have data in CSV or JSON format you can load it into an in-memory SQLite database and run queries against it directly in a single command using `sqlite-utils memory` like this:

```
$ sqlite-utils memory data.csv "select * from data"
```

You can pass multiple files to the command if you want to run joins between data from different files:

```
$ sqlite-utils memory one.csv two.json "select * from one join two on one.id = two.
↪other_id"
```

If your data is JSON it should be the same format supported by the *sqlite-utils insert command* - so either a single JSON object (treated as a single row) or a list of JSON objects.

CSV data can be comma- or tab- delimited.

The in-memory tables will be named after the files without their extensions. The tool also sets up aliases for those tables (using SQL views) as `t1`, `t2` and so on, or you can use the alias `t` to refer to the first table:

```
$ sqlite-utils memory example.csv "select * from t"
```

To read from standard input, use either `-` or `stdin` as the filename - then use `stdin` or `t` or `t1` as the table name:

```
$ cat example.csv | sqlite-utils memory - "select * from stdin"
```

Incoming CSV data will be assumed to use `utf-8`. If your data uses a different character encoding you can specify that with `--encoding`:

```
$ cat example.csv | sqlite-utils memory - "select * from stdin" --encoding=latin-1
```

If you are joining across multiple CSV files they must all use the same encoding.

Column types will be automatically detected in CSV or TSV data, using the same mechanism as `--detect-types` described in *Inserting CSV or TSV data*. You can pass the `--no-detect-types` option to disable this automatic type detection and treat all CSV and TSV columns as `TEXT`.

Explicitly specifying the format

By default, `sqlite-utils memory` will attempt to detect the incoming data format (JSON, TSV or CSV) automatically.

You can instead specify an explicit format by adding a `:csv`, `:tsv`, `:json` or `:nl` (for newline-delimited JSON) suffix to the filename. For example:

```
$ sqlite-utils memory one.dat:csv two.dat:nl "select * from one union select * from␣
↪two"
```

Here the contents of `one.dat` will be treated as CSV and the contents of `two.dat` will be treated as newline-delimited JSON.

To explicitly specify the format for data piped into the tool on standard input, use `stdin:format` - for example:

```
$ cat one.dat | sqlite-utils memory stdin:csv "select * from stdin"
```

Joining in-memory data against existing databases using –attach

The *attach option* can be used to attach database files to the in-memory connection, enabling joins between in-memory data loaded from a file and tables in existing SQLite database files. An example:

```
$ echo "id\n1\n3\n5" | sqlite-utils memory - --attach trees trees.db \
  "select * from trees.trees where rowid in (select id from stdin)"
```

Here the `--attach trees trees.db` option makes the `trees.db` database available with an alias of `trees`.

`select * from trees.trees where ...` can then query the `trees` table in that database.

The CSV data that was piped into the script is available in the `stdin` table, so `... where rowid in (select id from stdin)` can be used to return rows from the `trees` table that match IDs that were piped in as CSV content.

–schema, –dump and –save

To see the schema that will be created for a file or multiple files, use `--schema`:

```
% sqlite-utils memory dogs.csv --schema
CREATE TABLE [dogs] (
    [id] INTEGER,
    [age] INTEGER,
    [name] TEXT
);
CREATE VIEW t1 AS select * from [dogs];
CREATE VIEW t AS select * from [dogs];
```

You can output SQL that will both create the tables and insert the full data used to populate the in-memory database using `--dump`:

```
% sqlite-utils memory dogs.csv --dump
BEGIN TRANSACTION;
CREATE TABLE [dogs] (
    [id] INTEGER,
    [age] INTEGER,
    [name] TEXT
);
INSERT INTO "dogs" VALUES('1','4','Cleo');
INSERT INTO "dogs" VALUES('2','2','Pancakes');
CREATE VIEW t1 AS select * from [dogs];
CREATE VIEW t AS select * from [dogs];
COMMIT;
```

Passing `--save other.db` will instead use that SQL to populate a new database file:

```
% sqlite-utils memory dogs.csv --save dogs.db
```

These features are mainly intended as debugging tools - for much more finely grained control over how data is inserted into a SQLite database file see *Inserting JSON data* and *Inserting CSV or TSV data*.

1.2.3 Returning all rows in a table

You can return every row in a specified table using the `rows` command:

```
$ sqlite-utils rows dogs.db dogs
[{"id": 1, "age": 4, "name": "Cleo"},
 {"id": 2, "age": 2, "name": "Pancakes"}]
```

This command accepts the same output options as `query` - so you can pass `--nl`, `--csv`, `--tsv`, `--no-headers`, `--table` and `--fmt`.

You can use the `-c` option to specify a subset of columns to return:

```
$ sqlite-utils rows dogs.db dogs -c age -c name
[{"age": 4, "name": "Cleo"},
 {"age": 2, "name": "Pancakes"}]
```

1.2.4 Listing tables

You can list the names of tables in a database using the `tables` command:

```
$ sqlite-utils tables mydb.db
[{"table": "dogs"},
 {"table": "cats"},
 {"table": "chickens"}]
```

You can output this list in CSV using the `--csv` or `--tsv` options:

```
$ sqlite-utils tables mydb.db --csv --no-headers
dogs
cats
chickens
```

If you just want to see the FTS4 tables, you can use `--fts4` (or `--fts5` for FTS5 tables):

```
$ sqlite-utils tables docs.db --fts4
[{"table": "docs_fts"}]
```

Use `--counts` to include a count of the number of rows in each table:

```
$ sqlite-utils tables mydb.db --counts
[{"table": "dogs", "count": 12},
 {"table": "cats", "count": 332},
 {"table": "chickens", "count": 9}]
```

Use `--columns` to include a list of columns in each table:

```
$ sqlite-utils tables dogs.db --counts --columns
[{"table": "Gosh", "count": 0, "columns": ["c1", "c2", "c3"]},
 {"table": "Gosh2", "count": 0, "columns": ["c1", "c2", "c3"]},
 {"table": "dogs", "count": 2, "columns": ["id", "age", "name"]}]
```

Use `--schema` to include the schema of each table:

```
$ sqlite-utils tables dogs.db --schema --table
table    schema
-------  ----------------------------------------------
Gosh     CREATE TABLE Gosh (c1 text, c2 text, c3 text)
Gosh2    CREATE TABLE Gosh2 (c1 text, c2 text, c3 text)
dogs     CREATE TABLE [dogs] (
```

(continues on next page)

(continued from previous page)

```
            [id] INTEGER,
            [age] INTEGER,
            [name] TEXT)
```

The `--nl`, `--csv`, `--tsv`, `--table` and `--fmt` options are also available.

1.2.5 Listing views

The `views` command shows any views defined in the database:

```
$ sqlite-utils views sf-trees.db --table --counts --columns --schema
view         count  columns               schema
---------  -------  --------------------  ------------------------------------------
↪------------------
demo_view   189144  ['qSpecies']          CREATE VIEW demo_view AS select qSpecies␣
↪from Street_Tree_List
hello            1  ['sqlite_version()']  CREATE VIEW hello as select sqlite_version()
```

It takes the same options as the `tables` command:

- `--columns`
- `--schema`
- `--counts`
- `--nl`
- `--csv`
- `--tsv`
- `--table`

1.2.6 Listing indexes

The `indexes` command lists any indexes configured for the database:

```
$ sqlite-utils indexes covid.db --table
table                           index_name                                          ␣
↪      seqno    cid  name                 desc  coll      key
------------------------------  --------------------------------------------------
↪--  -------  -----  -----------------  ------  ------  -----
johns_hopkins_csse_daily_reports  idx_johns_hopkins_csse_daily_reports_combined_key  ␣
↪          0     12  combined_key            0  BINARY      1
johns_hopkins_csse_daily_reports  idx_johns_hopkins_csse_daily_reports_country_or_
↪region        0      1  country_or_region       0  BINARY      1
johns_hopkins_csse_daily_reports  idx_johns_hopkins_csse_daily_reports_province_or_
↪state         0      2  province_or_state       0  BINARY      1
johns_hopkins_csse_daily_reports  idx_johns_hopkins_csse_daily_reports_day           ␣
↪          0      0  day                     0  BINARY      1
ny_times_us_counties              idx_ny_times_us_counties_date                       ␣
↪          0      0  date                    1  BINARY      1
ny_times_us_counties              idx_ny_times_us_counties_fips                       ␣
↪          0      3  fips                    0  BINARY      1
```

(continues on next page)

(continued from previous page)

```
ny_times_us_counties              idx_ny_times_us_counties_county                 ␣
↪           0      1  county                    0  BINARY      1
ny_times_us_counties              idx_ny_times_us_counties_state                  ␣
↪           0      2  state                     0  BINARY      1
```

It shows indexes across all tables. To see indexes for specific tables, list those after the database:

```
$ sqlite-utils indexes covid.db johns_hopkins_csse_daily_reports --table
```

The command defaults to only showing the columns that are explicitly part of the index. To also include auxiliary columns use the `--aux` option - these columns will be listed with a `key` of `0`.

The command takes the same format options as the `tables` and `views` commands.

1.2.7 Listing triggers

The `triggers` command shows any triggers configured for the database:

```
$ sqlite-utils triggers global-power-plants.db --table
name             table      sql
---------------  ---------  ------------------------------------------------------------
↪-------
plants_insert    plants     CREATE TRIGGER [plants_insert] AFTER INSERT ON [plants]
                            BEGIN
                                INSERT OR REPLACE INTO [_counts]
                                VALUES (
                                  'plants',
                                  COALESCE(
                                    (SELECT count FROM [_counts] WHERE [table] =
↪'plants'),
                                  0
                                  ) + 1
                                );
                            END
```

It defaults to showing triggers for all tables. To see triggers for one or more specific tables pass their names as arguments:

```
$ sqlite-utils triggers global-power-plants.db plants
```

The command takes the same format options as the `tables` and `views` commands.

1.2.8 Showing the schema

The `sqlite-utils schema` command shows the full SQL schema for the database:

```
$ sqlite-utils schema dogs.db
CREATE TABLE "dogs" (
    [id] INTEGER PRIMARY KEY,
    [name] TEXT
);
```

This will show the schema for every table and index in the database. To view the schema just for a specified subset of tables pass those as additional arguments:

```
$ sqlite-utils schema dogs.db dogs chickens
...
```

1.2.9 Analyzing tables

When working with a new database it can be useful to get an idea of the shape of the data. The `sqlite-utils analyze-tables` command inspects specified tables (or all tables) and calculates some useful details about each of the columns in those tables.

To inspect the `tags` table in the `github.db` database, run the following:

```
$ sqlite-utils analyze-tables github.db tags
tags.repo: (1/3)

  Total rows: 261
  Null rows: 0
  Blank rows: 0

  Distinct values: 14

  Most common:
    88: 107914493
    75: 140912432
    27: 206156866

  Least common:
    1: 209590345
    2: 206649770
    2: 303218369

tags.name: (2/3)

  Total rows: 261
  Null rows: 0
  Blank rows: 0

  Distinct values: 175

  Most common:
    10: 0.2
    9: 0.1
    7: 0.3

  Least common:
    1: 0.1.1
    1: 0.11.1
    1: 0.1a2

tags.sha: (3/3)

  Total rows: 261
  Null rows: 0
  Blank rows: 0

  Distinct values: 261
```

For each column this tool displays the number of null rows, the number of blank rows (rows that contain an empty

string), the number of distinct values and, for columns that are not entirely distinct, the most common and least common values.

If you do not specify any tables every table in the database will be analyzed:

```
$ sqlite-utils analyze-tables github.db
```

If you wish to analyze one or more specific columns, use the `-c` option:

```
$ sqlite-utils analyze-tables github.db tags -c sha
```

Saving the analyzed table details

`analyze-tables` can take quite a while to run for large database files. You can save the results of the analysis to a database table called `_analyze_tables_` using the `--save` option:

```
$ sqlite-utils analyze-tables github.db --save
```

The `_analyze_tables_` table has the following schema:

```
CREATE TABLE [_analyze_tables_] (
    [table] TEXT,
    [column] TEXT,
    [total_rows] INTEGER,
    [num_null] INTEGER,
    [num_blank] INTEGER,
    [num_distinct] INTEGER,
    [most_common] TEXT,
    [least_common] TEXT,
    PRIMARY KEY ([table], [column])
);
```

The `most_common` and `least_common` columns will contain nested JSON arrays of the most common and least common values that look like this:

```
[
    ["Del Libertador, Av", 5068],
    ["Alberdi Juan Bautista Av.", 4612],
    ["Directorio Av.", 4552],
    ["Rivadavia, Av", 4532],
    ["Yerbal", 4512],
    ["Cosquín", 4472],
    ["Estado Plurinacional de Bolivia", 4440],
    ["Gordillo Timoteo", 4424],
    ["Montiel", 4360],
    ["Condarco", 4288]
]
```

1.2.10 Inserting JSON data

If you have data as JSON, you can use `sqlite-utils insert tablename` to insert it into a database. The table will be created with the correct (automatically detected) columns if it does not already exist.

You can pass in a single JSON object or a list of JSON objects, either as a filename or piped directly to standard-in (by using `-` as the filename).

Here's the simplest possible example:

```
$ echo '{"name": "Cleo", "age": 4}' | sqlite-utils insert dogs.db dogs -
```

To specify a column as the primary key, use `--pk=column_name`.

To create a compound primary key across more than one column, use `--pk` multiple times.

If you feed it a JSON list it will insert multiple records. For example, if `dogs.json` looks like this:

```
[
    {
        "id": 1,
        "name": "Cleo",
        "age": 4
    },
    {
        "id": 2,
        "name": "Pancakes",
        "age": 2
    },
    {
        "id": 3,
        "name": "Toby",
        "age": 6
    }
]
```

You can import all three records into an automatically created `dogs` table and set the `id` column as the primary key like so:

```
$ sqlite-utils insert dogs.db dogs dogs.json --pk=id
```

You can skip inserting any records that have a primary key that already exists using `--ignore`:

```
$ sqlite-utils insert dogs.db dogs dogs.json --ignore
```

You can delete all the existing rows in the table before inserting the new records using `--truncate`:

```
$ sqlite-utils insert dogs.db dogs dogs.json --truncate
```

Inserting binary data

You can insert binary data into a BLOB column by first encoding it using base64 and then structuring it like this:

```
[
    {
        "name": "transparent.gif",
        "content": {
            "$base64": true,
            "encoded": "R0lGODlhAQABAIAAAAAAAP///yH5BAEAAAAALAAAAAABAAEAAAIBRAA7"
        }
    }
]
```

Inserting newline-delimited JSON

You can also import newline-delimited JSON using the `--nl` option. Since Datasette can export newline-delimited JSON, you can combine the two tools like so:

```
$ curl -L "https://latest.datasette.io/fixtures/facetable.json?_shape=array&_nl=on" \
    | sqlite-utils insert nl-demo.db facetable - --pk=id --nl
```

This also means you pipe `sqlite-utils` together to easily create a new SQLite database file containing the results of a SQL query against another database:

```
$ sqlite-utils sf-trees.db \
    "select TreeID, qAddress, Latitude, Longitude from Street_Tree_List" --nl \
  | sqlite-utils insert saved.db trees - --nl
# This creates saved.db with a single table called trees:
$ sqlite-utils saved.db "select * from trees limit 5" --csv
TreeID,qAddress,Latitude,Longitude
141565,501X Baker St,37.7759676911831,-122.441396661871
232565,940 Elizabeth St,37.7517102172731,-122.441498017841
119263,495X Lakeshore Dr,,
207368,920 Kirkham St,37.760210314285,-122.47073935813
188702,1501 Evans Ave,37.7422086702947,-122.387293152263
```

Flattening nested JSON objects

`sqlite-utils insert` expects incoming data to consist of an array of JSON objects, where the top-level keys of each object will become columns in the created database table.

If your data is nested you can use the `--flatten` option to create columns that are derived from the nested data.

Consider this example document, in a file called `log.json`:

```
{
    "httpRequest": {
        "latency": "0.112114537s",
        "requestMethod": "GET",
        "requestSize": "534",
        "status": 200
    },
    "insertId": "6111722f000b5b4c4d4071e2",
    "labels": {
        "service": "datasette-io"
    }
}
```

Inserting this into a table using `sqlite-utils insert logs.db log log.json` will create a table with the following schema:

```
CREATE TABLE [logs] (
   [httpRequest] TEXT,
   [insertId] TEXT,
   [labels] TEXT
);
```

With the `--flatten` option columns will be created using `topkey_nextkey` column names - so running `sqlite-utils insert logs.db log log.json --flatten` will create the following schema instead:

```
CREATE TABLE [logs] (
   [httpRequest_latency] TEXT,
   [httpRequest_requestMethod] TEXT,
   [httpRequest_requestSize] TEXT,
   [httpRequest_status] INTEGER,
   [insertId] TEXT,
   [labels_service] TEXT
);
```

1.2.11 Inserting CSV or TSV data

If your data is in CSV format, you can insert it using the `--csv` option:

```
$ sqlite-utils insert dogs.db dogs docs.csv --csv
```

For tab-delimited data, use `--tsv`:

```
$ sqlite-utils insert dogs.db dogs dogs.tsv --tsv
```

Data is expected to be encoded as Unicode UTF-8. If your data is an another character encoding you can specify it using the `--encoding` option:

```
$ sqlite-utils insert dogs.db dogs dogs.tsv --tsv --encoding=latin-1
```

A progress bar is displayed when inserting data from a file. You can hide the progress bar using the `--silent` option.

By default every column inserted from a CSV or TSV file will be of type `TEXT`. To automatically detect column types - resulting in a mix of `TEXT`, `INTEGER` and `FLOAT` columns, use the `--detect-types` option (or its shortcut `-d`).

For example, given a `creatures.csv` file containing this:

```
name,age,weight
Cleo,6,45.5
Dori,1,3.5
```

The following command:

```
$ sqlite-utils insert creatures.db creatures creatures.csv --csv --detect-types
```

Will produce this schema:

```
$ sqlite-utils schema creatures.db
CREATE TABLE "creatures" (
   [name] TEXT,
   [age] INTEGER,
   [weight] FLOAT
);
```

You can set the `SQLITE_UTILS_DETECT_TYPES` environment variable if you want `--detect-types` to be the default behavior:

```
$ export SQLITE_UTILS_DETECT_TYPES=1
```

Alternative delimiters and quote characters

If your file uses a delimiter other than `,` or a quote character other than `"` you can attempt to detect delimiters or you can specify them explicitly.

The `--sniff` option can be used to attempt to detect the delimiters:

```
sqlite-utils insert dogs.db dogs dogs.csv --sniff
```

Alternatively, you can specify them using the `--delimiter` and `--quotechar` options.

Here's a CSV file that uses `;` for delimiters and the `|` symbol for quote characters:

```
name;description
Cleo;|Very fine; a friendly dog|
Pancakes;A local corgi
```

You can import that using:

```
$ sqlite-utils insert dogs.db dogs dogs.csv --delimiter=";" --quotechar="|"
```

Passing `--delimiter`, `--quotechar` or `--sniff` implies `--csv`, so you can omit the `--csv` option.

CSV files without a header row

The first row of any CSV or TSV file is expected to contain the names of the columns in that file.

If your file does not include this row, you can use the `--no-headers` option to specify that the tool should not use that fist row as headers.

If you do this, the table will be created with column names called `untitled_1` and `untitled_2` and so on. You can then rename them using the `sqlite-utils transform ... --rename` command, see *Transforming tables*.

1.2.12 Insert-replacing data

Insert-replacing works exactly like inserting, with the exception that if your data has a primary key that matches an already existing record that record will be replaced with the new data.

After running the above `dogs.json` example, try running this:

```
$ echo '{"id": 2, "name": "Pancakes", "age": 3}' | \
    sqlite-utils insert dogs.db dogs - --pk=id --replace
```

This will replace the record for id=2 (Pancakes) with a new record with an updated age.

1.2.13 Upserting data

Upserting is update-or-insert. If a row exists with the specified primary key the provided columns will be updated. If no row exists that row will be created.

Unlike `insert --replace`, an upsert will ignore any column values that exist but are not present in the upsert document.

For example:

```
$ echo '{"id": 2, "age": 4}' | \
    sqlite-utils upsert dogs.db dogs - --pk=id
```

This will update the dog with id=2 to have an age of 4, creating a new record (with a null name) if one does not exist. If a row DOES exist the name will be left as-is.

The command will fail if you reference columns that do not exist on the table. To automatically create missing columns, use the `--alter` option.

Note: `upsert` in sqlite-utils 1.x worked like `insert ... --replace` does in 2.x. See issue #66 for details of this change.

1.2.14 Inserting binary data from files

SQLite `BLOB` columns can be used to store binary content. It can be useful to insert the contents of files into a SQLite table.

The `insert-files` command can be used to insert the content of files, along with their metadata.

Here's an example that inserts all of the GIF files in the current directory into a `gifs.db` database, placing the file contents in an `images` table:

```
$ sqlite-utils insert-files gifs.db images *.gif
```

You can also pass one or more directories, in which case every file in those directories will be added recursively:

```
$ sqlite-utils insert-files gifs.db images path/to/my-gifs
```

By default this command will create a table with the following schema:

```
CREATE TABLE [images] (
    [path] TEXT PRIMARY KEY,
    [content] BLOB,
    [size] INTEGER
);
```

You can customize the schema using one or more `-c` options. For a table schema that includes just the path, MD5 hash and last modification time of the file, you would use this:

```
$ sqlite-utils insert-files gifs.db images *.gif -c path -c md5 -c mtime --pk=path
```

This will result in the following schema:

```
CREATE TABLE [images] (
    [path] TEXT PRIMARY KEY,
    [md5] TEXT,
    [mtime] FLOAT
);
```

You can change the name of one of these columns using a `-c colname:coldef` parameter. To rename the `mtime` column to `last_modified` you would use this:

```
$ sqlite-utils insert-files gifs.db images *.gif \
    -c path -c md5 -c last_modified:mtime --pk=path
```

You can pass `--replace` or `--upsert` to indicate what should happen if you try to insert a file with an existing primary key. Pass `--alter` to cause any missing columns to be added to the table.

The full list of column definitions you can use is as follows:

name The name of the file, e.g. `cleo.jpg`

path The path to the file relative to the root folder, e.g. `pictures/cleo.jpg`

fullpath The fully resolved path to the image, e.g. `/home/simonw/pictures/cleo.jpg`

sha256 The SHA256 hash of the file contents

md5 The MD5 hash of the file contents

mode The permission bits of the file, as an integer - you may want to convert this to octal

content The binary file contents, which will be stored as a BLOB

mtime The modification time of the file, as floating point seconds since the Unix epoch

ctime The creation time of the file, as floating point seconds since the Unix epoch

mtime_int The modification time as an integer rather than a float

ctime_int The creation time as an integer rather than a float

mtime_iso The modification time as an ISO timestamp, e.g. `2020-07-27T04:24:06.654246`

ctime_iso The creation time is an ISO timestamp

size The integer size of the file in bytes

You can insert data piped from standard input like this:

```
cat dog.jpg | sqlite-utils insert-files dogs.db pics - --name=dog.jpg
```

The `-` argument indicates data should be read from standard input. The string passed using the `--name` option will be used for the file name and path values.

When inserting data from standard input only the following column definitions are supported: `name`, `path`, `content`, `sha256`, `md5` and `size`.

1.2.15 Converting data in columns

The `convert` command can be used to transform the data in a specified column - for example to parse a date string into an ISO timestamp, or to split a string of tags into a JSON array.

The command accepts a database, table, one or more columns and a string of Python code to be executed against the values from those columns. The following example would replace the values in the `headline` column in the `articles` table with an upper-case version:

```
$ sqlite-utils convert content.db articles headline 'value.upper()'
```

The Python code is passed as a string. Within that Python code the `value` variable will be the value of the current column.

The code you provide will be compiled into a function that takes `value` as a single argument. If you break your function body into multiple lines the last line should be a `return` statement:

```
$ sqlite-utils convert content.db articles headline '
value = str(value)
return value.upper()'
```

You can specify Python modules that should be imported and made available to your code using one or more `--import` options. This example uses the `textwrap` module to wrap the `content` column at 100 characters:

```
$ sqlite-utils convert content.db articles content \
    '"\n".join(textwrap.wrap(value, 100))' \
    --import=textwrap
```

The transformation will be applied to every row in the specified table. You can limit that to just rows that match a `WHERE` clause using `--where`:

```
$ sqlite-utils convert content.db articles headline 'value.upper()' \
    --where "headline like '%cat%'"
```

You can include named parameters in your where clause and populate them using one or more `--param` options:

```
$ sqlite-utils convert content.db articles headline 'value.upper()' \
    --where "headline like :like" \
    --param like '%cat%'
```

The `--dry-run` option will output a preview of the conversion against the first ten rows, without modifying the database.

sqlite-utils convert recipes

Various built-in recipe functions are available for common operations. These are:

`r.jsonsplit(value, delimiter=',', type=<class 'str'>)` Convert a string like `a,b,c` into a JSON array `["a", "b", "c"]`

The `delimiter` parameter can be used to specify a different delimiter.

The `type` parameter can be set to `float` or `int` to produce a JSON array of different types, for example if the column's string value was `1.2,3,4.5` the following:

```
r.jsonsplit(value, type=float)
```

Would produce an array like this: `[1.2, 3.0, 4.5]`

`r.parsedate(value, dayfirst=False, yearfirst=False)` Parse a date and convert it to ISO date format: `yyyy-mm-dd`

In the case of dates such as `03/04/05` U.S. `MM/DD/YY` format is assumed - you can use `dayfirst=True` or `yearfirst=True` to change how these ambiguous dates are interpreted.

`r.parsedatetime(value, dayfirst=False, yearfirst=False)` Parse a datetime and convert it to ISO datetime format: `yyyy-mm-ddTHH:MM:SS`

These recipes can be used in the code passed to `sqlite-utils convert` like this:

```
$ sqlite-utils convert my.db mytable mycolumn \
  'r.jsonsplit(value)'
```

To use any of the documented parameters, do this:

```
$ sqlite-utils convert my.db mytable mycolumn \
  'r.jsonsplit(value, delimiter=":")'
```

Saving the result to a different column

The `--output` and `--output-type` options can be used to save the result of the conversion to a separate column, which will be created if that column does not already exist:

```
$ sqlite-utils convert content.db articles headline 'value.upper()' \
  --output headline_upper
```

The type of the created column defaults to `text`, but a different column type can be specified using `--output-type`. This example will create a new floating point column called `id_as_a_float` with a copy of each item's ID increased by 0.5:

```
$ sqlite-utils convert content.db articles id 'float(value) + 0.5' \
  --output id_as_a_float \
  --output-type float
```

You can drop the original column at the end of the operation by adding `--drop`.

Converting a column into multiple columns

Sometimes you may wish to convert a single column into multiple derived columns. For example, you may have a `location` column containing `latitude,longitude` values which you wish to split out into separate `latitude` and `longitude` columns.

You can achieve this using the `--multi` option to `sqlite-utils convert`. This option expects your Python code to return a Python dictionary: new columns well be created and populated for each of the keys in that dictionary.

For the `latitude,longitude` example you would use the following:

```
$ sqlite-utils convert demo.db places location \
'bits = value.split(",")
return {
  "latitude": float(bits[0]),
  "longitude": float(bits[1]),
}' --multi
```

The type of the returned values will be taken into account when creating the new columns. In this example, the resulting database schema will look like this:

```
CREATE TABLE [places] (
    [location] TEXT,
    [latitude] FLOAT,
    [longitude] FLOAT
);
```

The code function can also return `None`, in which case its output will be ignored. You can drop the original column at the end of the operation by adding `--drop`.

1.2.16 Creating tables

Most of the time creating tables by inserting example data is the quickest approach. If you need to create an empty table in advance of inserting data you can do so using the `create-table` command:

```
$ sqlite-utils create-table mydb.db mytable id integer name text --pk=id
```

This will create a table called `mytable` with two columns - an integer `id` column and a text `name` column. It will set the `id` column to be the primary key.

You can pass as many column-name column-type pairs as you like. Valid types are `integer`, `text`, `float` and `blob`.

You can specify columns that should be NOT NULL using `--not-null colname`. You can specify default values for columns using `--default colname defaultvalue`.

```
$ sqlite-utils create-table mydb.db mytable \
    id integer \
    name text \
    age integer \
    is_good integer \
    --not-null name \
    --not-null age \
    --default is_good 1 \
    --pk=id

$ sqlite-utils tables mydb.db --schema -t
table    schema
-------  --------------------------------
mytable  CREATE TABLE [mytable] (
            [id] INTEGER PRIMARY KEY,
            [name] TEXT NOT NULL,
            [age] INTEGER NOT NULL,
            [is_good] INTEGER DEFAULT '1'
        )
```

You can specify foreign key relationships between the tables you are creating using `--fk colname othertable othercolumn`:

```
$ sqlite-utils create-table books.db authors \
    id integer \
    name text \
    --pk=id

$ sqlite-utils create-table books.db books \
    id integer \
    title text \
    author_id integer \
    --pk=id \
    --fk author_id authors id

$ sqlite-utils tables books.db --schema -t
table    schema
-------  ------------------------------------------------
authors  CREATE TABLE [authors] (
            [id] INTEGER PRIMARY KEY,
            [name] TEXT
         )
books    CREATE TABLE [books] (
            [id] INTEGER PRIMARY KEY,
            [title] TEXT,
            [author_id] INTEGER REFERENCES [authors]([id])
         )
```

If a table with the same name already exists, you will get an error. You can choose to silently ignore this error with `--ignore`, or you can replace the existing table with a new, empty table using `--replace`.

1.2.17 Dropping tables

You can drop a table using the `drop-table` command:

```
$ sqlite-utils drop-table mydb.db mytable
```

Use `--ignore` to ignore the error if the table does not exist.

1.2.18 Transforming tables

The `transform` command allows you to apply complex transformations to a table that cannot be implemented using a regular SQLite `ALTER TABLE` command. See *Transforming a table* for details of how this works.

```
$ sqlite-utils transform mydb.db mytable \
    --drop column1 \
    --rename column2 column_renamed
```

Every option for this table (with the exception of `--pk-none`) can be specified multiple times. The options are as follows:

`--type column-name new-type` Change the type of the specified column. Valid types are `integer`, `text`, `float`, `blob`.

`--drop column-name` Drop the specified column.

`--rename column-name new-name` Rename this column to a new name.

`--column-order column` Use this multiple times to specify a new order for your columns. `-o` shortcut is also available.

`--not-null column-name` Set this column as `NOT NULL`.

`--not-null-false column-name` For a column that is currently set as `NOT NULL`, remove the `NOT NULL`.

`--pk column-name` Change the primary key column for this table. Pass `--pk` multiple times if you want to create a compound primary key.

`--pk-none` Remove the primary key from this table, turning it into a `rowid` table.

`--default column-name value` Set the default value of this column.

`--default-none column` Remove the default value for this column.

`--drop-foreign-key column` Drop the specified foreign key.

If you want to see the SQL that will be executed to make the change without actually executing it, add the `--sql` flag. For example:

```
$ sqlite-utils transform fixtures.db roadside_attractions \
    --rename pk id \
    --default name Untitled \
    --column-order id \
    --column-order longitude \
    --column-order latitude \
    --drop address \
    --sql
CREATE TABLE [roadside_attractions_new_4033a60276b9] (
   [id] INTEGER PRIMARY KEY,
   [longitude] FLOAT,
   [latitude] FLOAT,
```

(continues on next page)

(continued from previous page)

```
   [name] TEXT DEFAULT 'Untitled'
);
INSERT INTO [roadside_attractions_new_4033a60276b9] ([longitude], [latitude], [id],␣
↪[name])
   SELECT [longitude], [latitude], [pk], [name] FROM [roadside_attractions];
DROP TABLE [roadside_attractions];
ALTER TABLE [roadside_attractions_new_4033a60276b9] RENAME TO [roadside_attractions];
```

1.2.19 Extracting columns into a separate table

The `sqlite-utils extract` command can be used to extract specified columns into a separate table.

Take a look at the Python API documentation for *Extracting columns into a separate table* for a detailed description of how this works, including examples of table schemas before and after running an extraction operation.

The command takes a database, table and one or more columns that should be extracted. To extract the `species` column from the `trees` table you would run:

```
$ sqlite-utils extract my.db trees species
```

This would produce the following schema:

```
CREATE TABLE "trees" (
    [id] INTEGER PRIMARY KEY,
    [TreeAddress] TEXT,
    [species_id] INTEGER,
    FOREIGN KEY(species_id) REFERENCES species(id)
)

CREATE TABLE [species] (
    [id] INTEGER PRIMARY KEY,
    [species] TEXT
)
```

The command takes the following options:

`--table TEXT` The name of the lookup to extract columns to. This defaults to using the name of the columns that are being extracted.

`--fk-column TEXT` The name of the foreign key column to add to the table. Defaults to `columnname_id`.

`--rename <TEXT TEXT>` Use this option to rename the columns created in the new lookup table.

`--silent` Don't display the progress bar.

Here's a more complex example that makes use of these options. It converts this CSV file full of global power plants into SQLite, then extracts the `country` and `country_long` columns into a separate `countries` table:

```
wget 'https://github.com/wri/global-power-plant-database/blob/232a6666/output_
↪database/global_power_plant_database.csv?raw=true'
sqlite-utils insert global.db power_plants \
    'global_power_plant_database.csv?raw=true' --csv
# Extract those columns:
sqlite-utils extract global.db power_plants country country_long \
    --table countries \
    --fk-column country_id \
    --rename country_long name
```

After running the above, the command `sqlite-utils schema global.db` reveals the following schema:

```
CREATE TABLE [countries] (
   [id] INTEGER PRIMARY KEY,
   [country] TEXT,
   [name] TEXT
);
CREATE TABLE "power_plants" (
   [country_id] INTEGER,
   [name] TEXT,
   [gppd_idnr] TEXT,
   [capacity_mw] TEXT,
   [latitude] TEXT,
   [longitude] TEXT,
   [primary_fuel] TEXT,
   [other_fuel1] TEXT,
   [other_fuel2] TEXT,
   [other_fuel3] TEXT,
   [commissioning_year] TEXT,
   [owner] TEXT,
   [source] TEXT,
   [url] TEXT,
   [geolocation_source] TEXT,
   [wepp_id] TEXT,
   [year_of_capacity_data] TEXT,
   [generation_gwh_2013] TEXT,
   [generation_gwh_2014] TEXT,
   [generation_gwh_2015] TEXT,
   [generation_gwh_2016] TEXT,
   [generation_gwh_2017] TEXT,
   [generation_data_source] TEXT,
   [estimated_generation_gwh] TEXT,
   FOREIGN KEY([country_id]) REFERENCES [countries]([id])
);
CREATE UNIQUE INDEX [idx_countries_country_name]
    ON [countries] ([country], [name]);
```

1.2.20 Creating views

You can create a view using the `create-view` command:

```
$ sqlite-utils create-view mydb.db version "select sqlite_version()"

$ sqlite-utils mydb.db "select * from version"
[{"sqlite_version()": "3.31.1"}]
```

Use `--replace` to replace an existing view of the same name, and `--ignore` to do nothing if a view already exists.

1.2.21 Dropping views

You can drop a view using the `drop-view` command:

```
$ sqlite-utils drop-view myview
```

Use `--ignore` to ignore the error if the view does not exist.

1.2.22 Adding columns

You can add a column using the `add-column` command:

```
$ sqlite-utils add-column mydb.db mytable nameofcolumn text
```

The last argument here is the type of the column to be created. You can use one of `text`, `integer`, `float` or `blob`. If you leave it off, `text` will be used.

You can add a column that is a foreign key reference to another table using the `--fk` option:

```
$ sqlite-utils add-column mydb.db dogs species_id --fk species
```

This will automatically detect the name of the primary key on the species table and use that (and its type) for the new column.

You can explicitly specify the column you wish to reference using `--fk-col`:

```
$ sqlite-utils add-column mydb.db dogs species_id --fk species --fk-col ref
```

You can set a `NOT NULL DEFAULT 'x'` constraint on the new column using `--not-null-default`:

```
$ sqlite-utils add-column mydb.db dogs friends_count integer --not-null-default 0
```

1.2.23 Adding columns automatically on insert/update

You can use the `--alter` option to automatically add new columns if the data you are inserting or upserting is of a different shape:

```
$ sqlite-utils insert dogs.db dogs new-dogs.json --pk=id --alter
```

1.2.24 Adding foreign key constraints

The `add-foreign-key` command can be used to add new foreign key references to an existing table - something which SQLite's `ALTER TABLE` command does not support.

To add a foreign key constraint pointing the `books.author_id` column to `authors.id` in another table, do this:

```
$ sqlite-utils add-foreign-key books.db books author_id authors id
```

If you omit the other table and other column references `sqlite-utils` will attempt to guess them - so the above example could instead look like this:

```
$ sqlite-utils add-foreign-key books.db books author_id
```

Add `--ignore` to ignore an existing foreign key (as opposed to returning an error):

```
$ sqlite-utils add-foreign-key books.db books author_id --ignore
```

See *Adding foreign key constraints* in the Python API documentation for further details, including how the automatic table guessing mechanism works.

Adding multiple foreign keys at once

Adding a foreign key requires a `VACUUM`. On large databases this can be an expensive operation, so if you are adding multiple foreign keys you can combine them into one operation (and hence one `VACUUM`) using `add-foreign-keys`:

```
$ sqlite-utils add-foreign-keys books.db \
    books author_id authors id \
    authors country_id countries id
```

When you are using this command each foreign key needs to be defined in full, as four arguments - the table, column, other table and other column.

Adding indexes for all foreign keys

If you want to ensure that every foreign key column in your database has a corresponding index, you can do so like this:

```
$ sqlite-utils index-foreign-keys books.db
```

1.2.25 Setting defaults and not null constraints

You can use the `--not-null` and `--default` options (to both `insert` and `upsert`) to specify columns that should be `NOT NULL` or to set database defaults for one or more specific columns:

```
$ sqlite-utils insert dogs.db dogs_with_scores dogs-with-scores.json \
    --not-null=age \
    --not-null=name \
    --default age 2 \
    --default score 5
```

1.2.26 Creating indexes

You can add an index to an existing table using the `create-index` command:

```
$ sqlite-utils create-index mydb.db mytable col1 [col2...]
```

This can be used to create indexes against a single column or multiple columns.

The name of the index will be automatically derived from the table and columns. To specify a different name, use `--name=name_of_index`.

Use the `--unique` option to create a unique index.

Use `--if-not-exists` to avoid attempting to create the index if one with that name already exists.

To add an index on a column in descending order, prefix the column with a hyphen. Since this can be confused for a command-line option you need to construct that like this:

```
$ sqlite-utils create-index mydb.db mytable -- col1 -col2 col3
```

This will create an index on that table on `(col1, col2 desc, col3)`.

If your column names are already prefixed with a hyphen you'll need to manually execute a `CREATE INDEX` SQL statement to add indexes to them rather than using this tool.

1.2.27 Configuring full-text search

You can enable SQLite full-text search on a table and a set of columns like this:

```
$ sqlite-utils enable-fts mydb.db documents title summary
```

This will use SQLite's FTS5 module by default. Use `--fts4` if you want to use FTS4:

```
$ sqlite-utils enable-fts mydb.db documents title summary --fts4
```

The `enable-fts` command will populate the new index with all existing documents. If you later add more documents you will need to use `populate-fts` to cause them to be indexed as well:

```
$ sqlite-utils populate-fts mydb.db documents title summary
```

A better solution here is to use database triggers. You can set up database triggers to automatically update the full-text index using the `--create-triggers` option when you first run `enable-fts`:

```
$ sqlite-utils enable-fts mydb.db documents title summary --create-triggers
```

To set a custom FTS tokenizer, e.g. to enable Porter stemming, use `--tokenize=`:

```
$ sqlite-utils populate-fts mydb.db documents title summary --tokenize=porter
```

To remove the FTS tables and triggers you created, use `disable-fts`:

```
$ sqlite-utils disable-fts mydb.db documents
```

To rebuild one or more FTS tables (see *Rebuilding a full-text search table*), use `rebuild-fts`:

```
$ sqlite-utils rebuild-fts mydb.db documents
```

You can rebuild every FTS table by running `rebuild-fts` without passing any table names:

```
$ sqlite-utils rebuild-fts mydb.db
```

1.2.28 Executing searches

Once you have configured full-text search for a table, you can search it using `sqlite-utils search`:

```
$ sqlite-utils search mydb.db documents searchterm
```

This command accepts the same output options as `sqlite-utils query`: `--table`, `--csv`, `--tsv`, `--nl` etc.

By default it shows the most relevant matches first. You can specify a different sort order using the `-o` option, which can take a column or a column followed by `desc`:

```
# Sort by rowid
$ sqlite-utils search mydb.db documents searchterm -o rowid
# Sort by created in descending order
$ sqlite-utils search mydb.db documents searchterm -o 'created desc'
```

You can specify a subset of columns to be returned using the `-c` option one or more times:

```
$ sqlite-utils search mydb.db documents searchterm -c title -c created
```

By default all search results will be returned. You can use `--limit 20` to return just the first 20 results.

Use the `--sql` option to output the SQL that would be executed, rather than running the query:

```
$ sqlite-utils search mydb.db documents searchterm --sql
with original as (
    select
        rowid,
        *
    from [documents]
)
select
    [original].*
from
    [original]
    join [documents_fts] on [original].rowid = [documents_fts].rowid
where
    [documents_fts] match :query
order by
    [documents_fts].rank
```

1.2.29 Enabling cached counts

`select count(*)` queries can take a long time against large tables. `sqlite-utils` can speed these up by adding triggers to maintain a `_counts` table, see *Cached table counts using triggers* for details.

The `sqlite-utils enable-counts` command can be used to configure these triggers, either for every table in the database or for specific tables.

```
# Configure triggers for every table in the database
$ sqlite-utils enable-counts mydb.db

# Configure triggers just for specific tables
$ sqlite-utils enable-counts mydb.db table1 table2
```

If the `_counts` table ever becomes out-of-sync with the actual table counts you can repair it using the `reset-counts` command:

```
$ sqlite-utils reset-counts mydb.db
```

1.2.30 Vacuum

You can run VACUUM to optimize your database like so:

```
$ sqlite-utils vacuum mydb.db
```

1.2.31 Optimize

The optimize command can dramatically reduce the size of your database if you are using SQLite full-text search. It runs OPTIMIZE against all of your FTS4 and FTS5 tables, then runs VACUUM.

If you just want to run OPTIMIZE without the VACUUM, use the `--no-vacuum` flag.

```
# Optimize all FTS tables and then VACUUM
$ sqlite-utils optimize mydb.db

# Optimize but skip the VACUUM
$ sqlite-utils optimize --no-vacuum mydb.db
```

To optimize specific tables rather than every FTS table, pass those tables as extra arguments:

```
$ sqlite-utils optimize mydb.db table_1 table_2
```

1.2.32 WAL mode

You can enable Write-Ahead Logging for a database file using the `enable-wal` command:

```
$ sqlite-utils enable-wal mydb.db
```

You can disable WAL mode using `disable-wal`:

```
$ sqlite-utils disable-wal mydb.db
```

Both of these commands accept one or more database files as arguments.

1.2.33 Dumping the database to SQL

The `dump` command outputs a SQL dump of the schema and full contents of the specified database file:

```
$ sqlite-utils dump mydb.db
BEGIN TRANSACTION;
CREATE TABLE ...
...
COMMIT;
```

1.2.34 Loading SQLite extensions

Many of these commands have the ability to load additional SQLite extensions using the `--load-extension=/path/to/extension` option - use `--help` to check for support, e.g. `sqlite-utils rows --help`.

This option can be applied multiple times to load multiple extensions.

Since SpatiaLite is commonly used with SQLite, the value `spatialite` is special: it will search for SpatiaLite in the most common installation locations, saving you from needing to remember exactly where that module is located:

```
$ sqlite-utils memory "select spatialite_version()" --load-extension=spatialite
[{"spatialite_version()": "4.3.0a"}]
```

1.3 sqlite_utils Python library

1.3.1 Connecting to or creating a database

Database objects are constructed by passing in either a path to a file on disk or an existing SQLite3 database connection:

```
from sqlite_utils import Database

db = Database("my_database.db")
```

This will create `my_database.db` if it does not already exist.

If you want to recreate a database from scratch (first removing the existing file from disk if it already exists) you can use the `recreate=True` argument:

```
db = Database("my_database.db", recreate=True)
```

Instead of a file path you can pass in an existing SQLite connection:

```
import sqlite3

db = Database(sqlite3.connect("my_database.db"))
```

If you want to create an in-memory database, you can do so like this:

```
db = Database(memory=True)
```

Connections use `PRAGMA recursive_triggers=on` by default. If you don't want to use recursive triggers you can turn them off using:

```
db = Database(memory=True, recursive_triggers=False)
```

Attaching additional databases

SQLite supports cross-database SQL queries, which can join data from tables in more than one database file.

You can attach an additional database using the `.attach()` method, providing an alias to use for that database and the path to the SQLite file on disk.

```
db = Database("first.db")
db.attach("second", "second.db")
# Now you can run queries like this one:
print(db.query("""
select * from table_in_first
    union all
select * from second.table_in_second
"""))
```

You can reference tables in the attached database using the alias value you passed to `db.attach(alias, filepath)` as a prefix, for example the `second.table_in_second` reference in the SQL query above.

Tracing queries

You can use the `tracer` mechanism to see SQL queries that are being executed by SQLite. A tracer is a function that you provide which will be called with `sql` and `params` arguments every time SQL is executed, for example:

```
def tracer(sql, params):
    print("SQL: {} - params: {}".format(sql, params))
```

You can pass this function to the `Database()` constructor like so:

```
db = Database(memory=True, tracer=tracer)
```

You can also turn on a tracer function temporarily for a block of code using the `with db.tracer(...)` context manager:

```
db = Database(memory=True)
# ... later
with db.tracer(print):
    db["dogs"].insert({"name": "Cleo"})
```

This example will print queries only for the duration of the `with` block.

1.3.2 Executing queries

The `Database` class offers several methods for directly executing SQL queries.

db.query(sql, params)

The `db.query(sql)` function executes a SQL query and returns an iterator over Python dictionaries representing the resulting rows:

```
db = Database(memory=True)
db["dogs"].insert_all([{"name": "Cleo"}, {"name": "Pancakes"}])
for row in db.query("select * from dogs"):
    print(row)
```

(continues on next page)

(continued from previous page)

```
# Outputs:
# {'name': 'Cleo'}
# {'name': 'Pancakes'}
```

db.execute(sql, params)

The `db.execute()` and `db.executescript()` methods provide wrappers around `.execute()` and `.executescript()` on the underlying SQLite connection. These wrappers log to the *tracer function* if one has been registered.

`db.execute(sql)` returns a sqlite3.Cursor that was used to execute the SQL.

```
db = Database(memory=True)
db["dogs"].insert({"name": "Cleo"})
cursor = db.execute("update dogs set name = 'Cleopaws'")
print(cursor.rowcount)
# Outputs the number of rows affected by the update
# In this case 2
```

Other cursor methods such as `.fetchone()` and `.fetchall()` are also available, see the standard library documentation.

Passing parameters

Both `db.query()` and `db.execute()` accept an optional second argument for parameters to be passed to the SQL query.

This can take the form of either a tuple/list or a dictionary, depending on the type of parameters used in the query. Values passed in this way will be correctly quoted and escaped, helping avoid XSS vulnerabilities.

`?` parameters in the SQL query can be filled in using a list:

```
db.execute("update dogs set name = ?", ["Cleopaws"])
# This will rename ALL dogs to be called "Cleopaws"
```

Named parameters using `:name` can be filled using a dictionary:

```
dog = next(db.query(
    "select rowid, name from dogs where name = :name",
    {"name": "Cleopaws"}
))
# dog is now {'rowid': 1, 'name': 'Cleopaws'}
```

In this example `next()` is used to retrieve the first result in the iterator returned by the `db.query()` method.

1.3.3 Accessing tables

Tables are accessed using the indexing operator, like so:

```
table = db["my_table"]
```

If the table does not yet exist, it will be created the first time you attempt to insert or upsert data into it.

You can also access tables using the `.table()` method like so:

```
table = db.table("my_table")
```

Using this factory function allows you to set *Table configuration options*.

1.3.4 Listing tables

You can list the names of tables in a database using the `.table_names()` method:

```
>>> db.table_names()
['dogs']
```

To see just the FTS4 tables, use `.table_names(fts4=True)`. For FTS5, use `.table_names(fts5=True)`.

You can also iterate through the table objects themselves using the `.tables` property:

```
>>> db.tables
[<Table dogs>]
```

1.3.5 Listing views

`.view_names()` shows you a list of views in the database:

```
>>> db.view_names()
['good_dogs']
```

You can iterate through view objects using the `.views` property:

```
>>> db.views
[<View good_dogs>]
```

View objects are similar to Table objects, except that any attempts to insert or update data will throw an error. The full list of methods and properties available on a view object is as follows:

- `columns`
- `columns_dict`
- `count`
- `schema`
- `rows`
- `rows_where(where, where_args, order_by, select)`
- `drop()`

1.3.6 Listing rows

To iterate through dictionaries for each of the rows in a table, use `.rows`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> for row in db["dogs"].rows:
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
{'id': 2, 'age': 2, 'name': 'Pancakes'}
```

You can filter rows by a WHERE clause using `.rows_where(where, where_args)`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> for row in db["dogs"].rows_where("age > ?", [3]):
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
```

The first argument is a fragment of SQL. The second, optional argument is values to be passed to that fragment - you can use `?` placeholders and pass an array, or you can use `:named` parameters and pass a dictionary, like this:

```
>>> for row in db["dogs"].rows_where("age > :age", {"age": 3}):
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
```

To return custom columns (instead of the default that uses `select *`) pass `select="column1, column2"`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> for row in db["dogs"].rows_where(select='name, age'):
...     print(row)
{'name': 'Cleo', 'age': 4}
```

To specify an order, use the `order_by=` argument:

```
>>> for row in db["dogs"].rows_where("age > 1", order_by="age"):
...     print(row)
{'id': 2, 'age': 2, 'name': 'Pancakes'}
{'id': 1, 'age': 4, 'name': 'Cleo'}
```

You can use `order_by="age desc"` for descending order.

You can order all records in the table by excluding the `where` argument:

```
>>> for row in db["dogs"].rows_where(order_by="age desc"):
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
{'id': 2, 'age': 2, 'name': 'Pancakes'}
```

This method also accepts `offset=` and `limit=` arguments, for specifying an OFFSET and a LIMIT for the SQL query:

```
>>> for row in db["dogs"].rows_where(order_by="age desc", limit=1):
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
```

Counting rows

To count the number of rows that would be returned by a where filter, use `.count_where(where, where_args)`:

```
>>> db["dogs"].count_where("age > ?", [1]):
2
```

1.3.7 Listing rows with their primary keys

Sometimes it can be useful to retrieve the primary key along with each row, in order to pass that key (or primary key tuple) to the `.get()` or `.update()` methods.

The `.pks_and_rows_where()` method takes the same signature as `.rows_where()` (with the exception of the `select=` parameter) but returns a generator that yields pairs of `(primary key, row dictionary)`.

The primary key value will usually be a single value but can also be a tuple if the table has a compound primary key.

If the table is a `rowid` table (with no explicit primary key column) then that ID will be returned.

```
>>> db = sqlite_utils.Database(memory=True)
>>> db["dogs"].insert({"name": "Cleo"})
>>> for pk, row in db["dogs"].pks_and_rows_where():
...     print(pk, row)
1 {'rowid': 1, 'name': 'Cleo'}

>>> db["dogs_with_pk"].insert({"id": 5, "name": "Cleo"}, pk="id")
>>> for pk, row in db["dogs_with_pk"].pks_and_rows_where():
...     print(pk, row)
5 {'id': 5, 'name': 'Cleo'}

>>> db["dogs_with_compound_pk"].insert(
...     {"species": "dog", "id": 3, "name": "Cleo"},
...     pk=("species", "id")
... )
>>> for pk, row in db["dogs_with_compound_pk"].pks_and_rows_where():
...     print(pk, row)
('dog', 3) {'species': 'dog', 'id': 3, 'name': 'Cleo'}
```

1.3.8 Retrieving a specific record

You can retrieve a record by its primary key using `table.get()`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> print(db["dogs"].get(1))
{'id': 1, 'age': 4, 'name': 'Cleo'}
```

If the table has a compound primary key you can pass in the primary key values as a tuple:

```
>>> db["compound_dogs"].get(("mixed", 3))
```

If the record does not exist a `NotFoundError` will be raised:

```
from sqlite_utils.db import NotFoundError

try:
    row = db["dogs"].get(5)
except NotFoundError:
    print("Dog not found")
```

1.3.9 Showing the schema

The `db.schema` property returns the full SQL schema for the database as a string:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> print(db.schema)
>>> print(db.schema)
CREATE TABLE "dogs" (
```

(continues on next page)

(continued from previous page)

```
    [id] INTEGER PRIMARY KEY,
    [name] TEXT
);
```

1.3.10 Creating tables

The easiest way to create a new table is to insert a record into it:

```
from sqlite_utils import Database
import sqlite3

db = Database(sqlite3.connect("/tmp/dogs.db"))
dogs = db["dogs"]
dogs.insert({
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 3,
    "is_good_dog": True,
})
```

This will automatically create a new table called "dogs" with the following schema:

```
CREATE TABLE dogs (
    name TEXT,
    twitter TEXT,
    age INTEGER,
    is_good_dog INTEGER
)
```

You can also specify a primary key by passing the `pk=` parameter to the `.insert()` call. This will only be obeyed if the record being inserted causes the table to be created:

```
dogs.insert({
    "id": 1,
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 3,
    "is_good_dog": True,
}, pk="id")
```

After inserting a row like this, the `dogs.last_rowid` property will return the SQLite `rowid` assigned to the most recently inserted record.

The `dogs.last_pk` property will return the last inserted primary key value, if you specified one. This can be very useful when writing code that creates foreign keys or many-to-many relationships.

Custom column order and column types

The order of the columns in the table will be derived from the order of the keys in the dictionary, provided you are using Python 3.6 or later.

If you want to explicitly set the order of the columns you can do so using the `column_order=` parameter:

```
db["dogs"].insert({
    "id": 1,
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 3,
    "is_good_dog": True,
}, pk="id", column_order=("id", "twitter", "name"))
```

You don't need to pass all of the columns to the `column_order` parameter. If you only pass a subset of the columns the remaining columns will be ordered based on the key order of the dictionary.

Column types are detected based on the example data provided. Sometimes you may find you need to over-ride these detected types - to create an integer column for data that was provided as a string for example, or to ensure that a table where the first example was `None` is created as an `INTEGER` rather than a `TEXT` column. You can do this using the `columns=` parameter:

```
db["dogs"].insert({
    "id": 1,
    "name": "Cleo",
    "age": "5",
}, pk="id", columns={"age": int, "weight": float})
```

This will create a table with the following schema:

```
CREATE TABLE [dogs] (
    [id] INTEGER PRIMARY KEY,
    [name] TEXT,
    [age] INTEGER,
    [weight] FLOAT
)
```

Explicitly creating a table

You can directly create a new table without inserting any data into it using the `.create()` method:

```
db["cats"].create({
    "id": int,
    "name": str,
    "weight": float,
}, pk="id")
```

The first argument here is a dictionary specifying the columns you would like to create. Each column is paired with a Python type indicating the type of column. See *Adding columns* for full details on how these types work.

This method takes optional arguments `pk=`, `column_order=`, `foreign_keys=`, `not_null=set()` and `defaults=dict()` - explained below.

Compound primary keys

If you want to create a table with a compound primary key that spans multiple columns, you can do so by passing a tuple of column names to any of the methods that accept a `pk=` parameter. For example:

```
db["cats"].create({
    "id": int,
    "breed": str,
```

(continues on next page)

(continued from previous page)

```
    "name": str,
    "weight": float,
}, pk=("breed", "id"))
```

This also works for the `.insert()`, `.insert_all()`, `.upsert()` and `.upsert_all()` methods.

Specifying foreign keys

Any operation that can create a table (`.create()`, `.insert()`, `.insert_all()`, `.upsert()` and `.upsert_all()`) accepts an optional `foreign_keys=` argument which can be used to set up foreign key constraints for the table that is being created.

If you are using your database with Datasette, Datasette will detect these constraints and use them to generate hyperlinks to associated records.

The `foreign_keys` argument takes a list that indicates which foreign keys should be created. The list can take several forms. The simplest is a list of columns:

```
foreign_keys=["author_id"]
```

The library will guess which tables you wish to reference based on the column names using the rules described in *Adding foreign key constraints*.

You can also be more explicit, by passing in a list of tuples:

```
foreign_keys=[
    ("author_id", "authors", "id")
]
```

This means that the `author_id` column should be a foreign key that references the `id` column in the `authors` table.

You can leave off the third item in the tuple to have the referenced column automatically set to the primary key of that table. A full example:

```
db["authors"].insert_all([
    {"id": 1, "name": "Sally"},
    {"id": 2, "name": "Asheesh"}
], pk="id")
db["books"].insert_all([
    {"title": "Hedgehogs of the world", "author_id": 1},
    {"title": "How to train your wolf", "author_id": 2},
], foreign_keys=[
    ("author_id", "authors")
])
```

1.3.11 Table configuration options

The `.insert()`, `.upsert()`, `.insert_all()` and `.upsert_all()` methods each take a number of keyword arguments, some of which influence what happens should they cause a table to be created and some of which affect the behavior of those methods.

You can set default values for these methods by accessing the table through the `db.table(...)` method (instead of using `db["table_name"]`), like so:

```
table = db.table(
    "authors",
    pk="id",
    not_null={"name", "score"},
    column_order=("id", "name", "score", "url")
)
# Now you can call .insert() like so:
table.insert({"id": 1, "name": "Tracy", "score": 5})
```

The configuration options that can be specified in this way are `pk`, `foreign_keys`, `column_order`, `not_null`, `defaults`, `batch_size`, `hash_id`, `alter`, `ignore`, `replace`, `extracts`, `conversions`, `columns`. These are all documented below.

1.3.12 Setting defaults and not null constraints

Each of the methods that can cause a table to be created take optional arguments `not_null=set()` and `defaults=dict()`. The methods that take these optional arguments are:

- `db.create_table(...)`
- `table.create(...)`
- `table.insert(...)`
- `table.insert_all(...)`
- `table.upsert(...)`
- `table.upsert_all(...)`

You can use `not_null=` to pass a set of column names that should have a `NOT NULL` constraint set on them when they are created.

You can use `defaults=` to pass a dictionary mapping columns to the default value that should be specified in the `CREATE TABLE` statement.

Here's an example that uses these features:

```
db["authors"].insert_all(
    [{"id": 1, "name": "Sally", "score": 2}],
    pk="id",
    not_null={"name", "score"},
    defaults={"score": 1},
)
db["authors"].insert({"name": "Dharma"})

list(db["authors"].rows)
# Outputs:
# [{'id': 1, 'name': 'Sally', 'score': 2},
#  {'id': 3, 'name': 'Dharma', 'score': 1}]
print(db["authors"].schema)
# Outputs:
# CREATE TABLE [authors] (
#     [id] INTEGER PRIMARY KEY,
#     [name] TEXT NOT NULL,
#     [score] INTEGER NOT NULL DEFAULT 1
# )
```

1.3.13 Bulk inserts

If you have more than one record to insert, the `insert_all()` method is a much more efficient way of inserting them. Just like `insert()` it will automatically detect the columns that should be created, but it will inspect the first batch of 100 items to help decide what those column types should be.

Use it like this:

```
db["dogs"].insert_all([{
    "id": 1,
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 3,
    "is_good_dog": True,
}, {
    "id": 2,
    "name": "Marnie",
    "twitter": "MarnieTheDog",
    "age": 16,
    "is_good_dog": True,
}], pk="id", column_order=("id", "twitter", "name"))
```

The column types used in the `CREATE TABLE` statement are automatically derived from the types of data in that first batch of rows. Any additional columns in subsequent batches will cause a `sqlite3.OperationalError` exception to be raised unless the `alter=True` argument is supplied, in which case the new columns will be created.

The function can accept an iterator or generator of rows and will commit them according to the batch size. The default batch size is 100, but you can specify a different size using the `batch_size` parameter:

```
db["big_table"].insert_all(({
    "id": 1,
    "name": "Name {}".format(i),
} for i in range(10000)), batch_size=1000)
```

You can skip inserting any records that have a primary key that already exists using `ignore=True`. This works with both `.insert({...}, ignore=True)` and `.insert_all([...], ignore=True)`.

You can delete all the existing rows in the table before inserting the new records using `truncate=True`. This is useful if you want to replace the data in the table.

1.3.14 Insert-replacing data

If you want to insert a record or replace an existing record with the same primary key, using the `replace=True` argument to `.insert()` or `.insert_all()`:

```
db["dogs"].insert_all([{
    "id": 1,
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 3,
    "is_good_dog": True,
}, {
    "id": 2,
    "name": "Marnie",
    "twitter": "MarnieTheDog",
    "age": 16,
```

(continues on next page)

(continued from previous page)

```
    "is_good_dog": True,
}], pk="id", replace=True)
```

Note: Prior to sqlite-utils 2.x the `.upsert()` and `.upsert_all()` methods did this. See *Upserting data* for the new behaviour of those methods in 2.x.

1.3.15 Updating a specific record

You can update a record by its primary key using `table.update()`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> print(db["dogs"].get(1))
{'id': 1, 'age': 4, 'name': 'Cleo'}
>>> db["dogs"].update(1, {"age": 5})
>>> print(db["dogs"].get(1))
{'id': 1, 'age': 5, 'name': 'Cleo'}
```

The first argument to `update()` is the primary key. This can be a single value, or a tuple if that table has a compound primary key:

```
>>> db["compound_dogs"].update((5, 3), {"name": "Updated"})
```

The second argument is a dictionary of columns that should be updated, along with their new values.

You can cause any missing columns to be added automatically using `alter=True`:

```
>>> db["dogs"].update(1, {"breed": "Mutt"}, alter=True)
```

1.3.16 Deleting a specific record

You can delete a record using `table.delete()`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> db["dogs"].delete(1)
```

The `delete()` method takes the primary key of the record. This can be a tuple of values if the row has a compound primary key:

```
>>> db["compound_dogs"].delete((5, 3))
```

1.3.17 Deleting multiple records

You can delete all records in a table that match a specific WHERE statement using `table.delete_where()`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> # Delete every dog with age less than 3
>>> db["dogs"].delete_where("age < ?", [3])
```

Calling `table.delete_where()` with no other arguments will delete every row in the table.

1.3.18 Upserting data

Upserting allows you to insert records if they do not exist and update them if they DO exist, based on matching against their primary key.

For example, given the dogs database you could upsert the record for Cleo like so:

```
db["dogs"].upsert({
    "id": 1,
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 4,
    "is_good_dog": True,
}, pk="id", column_order=("id", "twitter", "name"))
```

If a record exists with id=1, it will be updated to match those fields. If it does not exist it will be created.

Any existing columns that are not referenced in the dictionary passed to `.upsert()` will be unchanged. If you want to replace a record entirely, use `.insert(doc, replace=True)` instead.

Note that the `pk` and `column_order` parameters here are optional if you are certain that the table has already been created. You should pass them if the table may not exist at the time the first upsert is performed.

An `upsert_all()` method is also available, which behaves like `insert_all()` but performs upserts instead.

Note: `.upsert()` and `.upsert_all()` in sqlite-utils 1.x worked like `.insert(..., replace=True)` and `.insert_all(..., replace=True)` do in 2.x. See issue #66 for details of this change.

1.3.19 Converting data in columns

The `table.convert(...)` method can be used to apply a conversion function to the values in a column, either to update that column or to populate new columns. It is the Python library equivalent of the *sqlite-utils convert* command.

This feature works by registering a custom SQLite function that applies a Python transformation, then running a SQL query equivalent to `UPDATE table SET column = convert_value(column);`

To transform a specific column to uppercase, you would use the following:

```
db["dogs"].convert("name", lambda value: value.upper())
```

You can pass a list of columns, in which case the transformation will be applied to each one:

```
db["dogs"].convert(["name", "twitter"], lambda value: value.upper())
```

To save the output to of the transformation to a different column, use the `output=` parameter:

```
db["dogs"].convert("name", lambda value: value.upper(), output="name_upper")
```

This will add the new column, if it does not already exist. You can pass `output_type=int` or some other type to control the type of the new column - otherwise it will default to text.

If you want to drop the original column after saving the results in a separate output column, pass `drop=True`.

You can create multiple new columns from a single input column by passing `multi=True` and a conversion function that returns a Python dictionary. This example creates new `upper` and `lower` columns populated from the single `title` column:

```
table.convert(
    "title", lambda v: {"upper": v.upper(), "lower": v.lower()}, multi=True
)
```

The `.convert()` method accepts optional `where=` and `where_args=` parameters which can be used to apply the conversion to a subset of rows specified by a where clause. Here's how to apply the conversion only to rows with an `id` that is higher than 20:

```
table.convert("title", lambda v: v.upper(), where="id > :id", where_args={"id": 20})
```

These behave the same as the corresponding parameters to the *.rows_where()* method, so you can use `?` placeholders and a list of values instead of `:named` placeholders with a dictionary.

1.3.20 Working with lookup tables

A useful pattern when populating large tables in to break common values out into lookup tables. Consider a table of `Trees`, where each tree has a species. Ideally these species would be split out into a separate `Species` table, with each one assigned an integer primary key that can be referenced from the `Trees` table `species_id` column.

Creating lookup tables explicitly

Calling `db["Species"].lookup({"name": "Palm"})` creates a table called `Species` (if one does not already exist) with two columns: `id` and `name`. It sets up a unique constraint on the `name` column to guarantee it will not contain duplicate rows. It then inserts a new row with the `name` set to `Palm` and returns the new integer primary key value.

If the `Species` table already exists, it will insert the new row and return the primary key. If a row with that `name` already exists, it will return the corresponding primary key value directly.

If you call `.lookup()` against an existing table without the unique constraint it will attempt to add the constraint, raising an `IntegrityError` if the constraint cannot be created.

If you pass in a dictionary with multiple values, both values will be used to insert or retrieve the corresponding ID and any unique constraint that is created will cover all of those columns, for example:

```
db["Trees"].insert({
    "latitude": 49.1265976,
    "longitude": 2.5496218,
    "species": db["Species"].lookup({
        "common_name": "Common Juniper",
        "latin_name": "Juniperus communis"
    })
})
```

Populating lookup tables automatically during insert/upsert

A more efficient way to work with lookup tables is to define them using the `extracts=` parameter, which is accepted by `.insert()`, `.upsert()`, `.insert_all()`, `.upsert_all()` and by the `.table(...)` factory function.

`extracts=` specifies columns which should be "extracted" out into a separate lookup table during the data insertion.

It can be either a list of column names, in which case the extracted table names will match the column names exactly, or it can be a dictionary mapping column names to the desired name of the extracted table.

To extract the `species` column out to a separate `Species` table, you can do this:

```
# Using the table factory
trees = db.table("Trees", extracts={"species": "Species"})
trees.insert({
    "latitude": 49.1265976,
    "longitude": 2.5496218,
    "species": "Common Juniper"
})

# If you want the table to be called 'species', you can do this:
trees = db.table("Trees", extracts=["species"])

# Using .insert() directly
db["Trees"].insert({
    "latitude": 49.1265976,
    "longitude": 2.5496218,
    "species": "Common Juniper"
}, extracts={"species": "Species"})
```

1.3.21 Working with many-to-many relationships

`sqlite-utils` includes a shortcut for creating records using many-to-many relationships in the form of the `table.m2m(...)` method.

Here's how to create two new records and connect them via a many-to-many table in a single line of code:

```
db["dogs"].insert({"id": 1, "name": "Cleo"}, pk="id").m2m(
    "humans", {"id": 1, "name": "Natalie"}, pk="id"
)
```

Running this example actually creates three tables: `dogs`, `humans` and a many-to-many `dogs_humans` table. It will insert a record into each of those tables.

The `.m2m()` method executes against the last record that was affected by `.insert()` or `.update()` - the record identified by the `table.last_pk` property. To execute `.m2m()` against a specific record you can first select it by passing its primary key to `.update()`:

```
db["dogs"].update(1).m2m(
    "humans", {"id": 2, "name": "Simon"}, pk="id"
)
```

The first argument to `.m2m()` can be either the name of a table as a string or it can be the table object itself.

The second argument can be a single dictionary record or a list of dictionaries. These dictionaries will be passed to `.upsert()` against the specified table.

Here's alternative code that creates the dog record and adds two people to it:

```
db = Database(memory=True)
dogs = db.table("dogs", pk="id")
humans = db.table("humans", pk="id")
dogs.insert({"id": 1, "name": "Cleo"}).m2m(
    humans, [
        {"id": 1, "name": "Natalie"},
        {"id": 2, "name": "Simon"}
    ]
)
```

The method will attempt to find an existing many-to-many table by looking for a table that has foreign key relationships against both of the tables in the relationship.

If it cannot find such a table, it will create a new one using the names of the two tables - `dogs_humans` in this example. You can customize the name of this table using the `m2m_table=` argument to `.m2m()`.

It it finds multiple candidate tables with foreign keys to both of the specified tables it will raise a `sqlite_utils.db.NoObviousTable` exception. You can avoid this error by specifying the correct table using `m2m_table=`.

The `.m2m()` method also takes an optional `pk=` argument to specify the primary key that should be used if the table is created, and an optional `alter=True` argument to specify that any missing columns of an existing table should be added if they are needed.

Using m2m and lookup tables together

You can work with (or create) lookup tables as part of a call to `.m2m()` using the `lookup=` parameter. This accepts the same argument as `table.lookup()` does - a dictionary of values that should be used to lookup or create a row in the lookup table.

This example creates a dogs table, populates it, creates a characteristics table, populates that and sets up a many-to-many relationship between the two. It chains `.m2m()` twice to create two associated characteristics:

```
db = Database(memory=True)
dogs = db.table("dogs", pk="id")
dogs.insert({"id": 1, "name": "Cleo"}).m2m(
    "characteristics", lookup={
        "name": "Playful"
    }
).m2m(
    "characteristics", lookup={
        "name": "Opinionated"
    }
)
```

You can inspect the database to see the results like this:

```
>>> db.table_names()
['dogs', 'characteristics', 'characteristics_dogs']
>>> list(db["dogs"].rows)
[{'id': 1, 'name': 'Cleo'}]
>>> list(db["characteristics"].rows)
[{'id': 1, 'name': 'Playful'}, {'id': 2, 'name': 'Opinionated'}]
>>> list(db["characteristics_dogs"].rows)
[{'characteristics_id': 1, 'dogs_id': 1}, {'characteristics_id': 2, 'dogs_id': 1}]
>>> print(db["characteristics_dogs"].schema)
CREATE TABLE [characteristics_dogs] (
    [characteristics_id] INTEGER REFERENCES [characteristics]([id]),
    [dogs_id] INTEGER REFERENCES [dogs]([id]),
    PRIMARY KEY ([characteristics_id], [dogs_id])
)
```

1.3.22 Analyzing a column

The `table.analyze_column(column, common_limit=10, value_truncate=None)` method is used by the *analyze-tables* CLI command. It returns a `ColumnDetails` named tuple with the following fields:

`table` The name of the table

column The name of the column

total_rows The total number of rows in the table‘

num_null The number of rows for which this column is null

num_blank The number of rows for which this column is blank (the empty string)

num_distinct The number of distinct values in this column

most_common The `N` most common values as a list of `(value, count)` tuples‘, or `None` if the table consists entirely of distinct values

least_common The `N` least common values as a list of `(value, count)` tuples‘, or `None` if the table is entirely distinct or if the number of distinct values is less than N (since they will already have been returned in `most_common`)

`N` defaults to 10, or you can pass a custom `N` using the `common_limit` parameter.

You can use the `value_truncate` parameter to truncate values in the `most_common` and `least_common` lists to a specified number of characters.

1.3.23 Adding columns

You can add a new column to a table using the `.add_column(col_name, col_type)` method:

```
db["dogs"].add_column("instagram", str)
db["dogs"].add_column("weight", float)
db["dogs"].add_column("dob", datetime.date)
db["dogs"].add_column("image", "BLOB")
db["dogs"].add_column("website") # str by default
```

You can specify the `col_type` argument either using a SQLite type as a string, or by directly passing a Python type e.g. `str` or `float`.

The `col_type` is optional - if you omit it the type of `TEXT` will be used.

SQLite types you can specify are `"TEXT"`, `"INTEGER"`, `"FLOAT"` or `"BLOB"`.

If you pass a Python type, it will be mapped to SQLite types as shown here:

```
float: "FLOAT"
int: "INTEGER"
bool: "INTEGER"
str: "TEXT"
bytes: "BLOB"
datetime.datetime: "TEXT"
datetime.date: "TEXT"
datetime.time: "TEXT"

# If numpy is installed
np.int8: "INTEGER"
np.int16: "INTEGER"
np.int32: "INTEGER"
np.int64: "INTEGER"
np.uint8: "INTEGER"
np.uint16: "INTEGER"
np.uint32: "INTEGER"
np.uint64: "INTEGER"
np.float16: "FLOAT"
```

(continues on next page)

(continued from previous page)

```
np.float32: "FLOAT"
np.float64: "FLOAT"
```

You can also add a column that is a foreign key reference to another table using the `fk` parameter:

```
db["dogs"].add_column("species_id", fk="species")
```

This will automatically detect the name of the primary key on the species table and use that (and its type) for the new column.

You can explicitly specify the column you wish to reference using `fk_col`:

```
db["dogs"].add_column("species_id", fk="species", fk_col="ref")
```

You can set a `NOT NULL DEFAULT 'x'` constraint on the new column using `not_null_default`:

```
db["dogs"].add_column("friends_count", int, not_null_default=0)
```

1.3.24 Adding columns automatically on insert/update

You can insert or update data that includes new columns and have the table automatically altered to fit the new schema using the `alter=True` argument. This can be passed to all four of `.insert()`, `.upsert()`, `.insert_all()` and `.upsert_all()`, or it can be passed to `db.table(table_name, alter=True)` to enable it by default for all method calls against that table instance.

```
db["new_table"].insert({"name": "Gareth"})
# This will throw an exception:
db["new_table"].insert({"name": "Gareth", "age": 32})
# This will succeed and add a new "age" integer column:
db["new_table"].insert({"name": "Gareth", "age": 32}, alter=True)
# You can see confirm the new column like so:
print(db["new_table"].columns_dict)
# Outputs this:
# {'name': <class 'str'>, 'age': <class 'int'>}

# This works too:
new_table = db.table("new_table", alter=True)
new_table.insert({"name": "Gareth", "age": 32, "shoe_size": 11})
```

1.3.25 Adding foreign key constraints

The SQLite `ALTER TABLE` statement doesn't have the ability to add foreign key references to an existing column.

It's possible to add these references through very careful manipulation of SQLite's `sqlite_master` table, using `PRAGMA writable_schema`.

`sqlite-utils` can do this for you, though there is a significant risk of data corruption if something goes wrong so it is advisable to create a fresh copy of your database file before attempting this.

Here's an example of this mechanism in action:

```
db["authors"].insert_all([
    {"id": 1, "name": "Sally"},
    {"id": 2, "name": "Asheesh"}
```

(continues on next page)

(continued from previous page)

```
], pk="id")
db["books"].insert_all([
    {"title": "Hedgehogs of the world", "author_id": 1},
    {"title": "How to train your wolf", "author_id": 2},
])
db["books"].add_foreign_key("author_id", "authors", "id")
```

The `table.add_foreign_key(column, other_table, other_column)` method takes the name of the column, the table that is being referenced and the key column within that other table. If you omit the `other_column` argument the primary key from that table will be used automatically. If you omit the `other_table` argument the table will be guessed based on some simple rules:

- If the column is of format `author_id`, look for tables called `author` or `authors`
- If the column does not end in `_id`, try looking for a table with the exact name of the column or that name with an added `s`

This method first checks that the specified foreign key references tables and columns that exist and does not clash with an existing foreign key. It will raise a `sqlite_utils.db.AlterError` exception if these checks fail.

To ignore the case where the key already exists, use `ignore=True`:

```
db["books"].add_foreign_key("author_id", "authors", "id", ignore=True)
```

Adding multiple foreign key constraints at once

The final step in adding a new foreign key to a SQLite database is to run `VACUUM`, to ensure the new foreign key is available in future introspection queries.

`VACUUM` against a large (multi-GB) database can take several minutes or longer. If you are adding multiple foreign keys using `table.add_foreign_key(...)` these can quickly add up.

Instead, you can use `db.add_foreign_keys(...)` to add multiple foreign keys within a single transaction. This method takes a list of four-tuples, each one specifying a `table`, `column`, `other_table` and `other_column`.

Here's an example adding two foreign keys at once:

```
db.add_foreign_keys([
    ("dogs", "breed_id", "breeds", "id"),
    ("dogs", "home_town_id", "towns", "id")
])
```

This method runs the same checks as `.add_foreign_keys()` and will raise `sqlite_utils.db.AlterError` if those checks fail.

Adding indexes for all foreign keys

If you want to ensure that every foreign key column in your database has a corresponding index, you can do so like this:

```
db.index_foreign_keys()
```

1.3.26 Dropping a table or view

You can drop a table or view using the `.drop()` method:

```
db["my_table"].drop()
```

Pass `ignore=True` if you want to ignore the error caused by the table or view not existing.

```
db["my_table"].drop(ignore=True)
```

1.3.27 Transforming a table

The SQLite `ALTER TABLE` statement is limited. It can add columns and rename tables, but it cannot drop columns, change column types, change `NOT NULL` status or change the primary key for a table.

The `table.transform()` method can do all of these things, by implementing a multi-step pattern described in the SQLite documentation:

1. Start a transaction
2. `CREATE TABLE tablename_new_x123` with the required changes
3. Copy the old data into the new table using `INSERT INTO tablename_new_x123 SELECT * FROM tablename;`
4. `DROP TABLE tablename;`
5. `ALTER TABLE tablename_new_x123 RENAME TO tablename;`
6. Commit the transaction

The `.transform()` method takes a number of parameters, all of which are optional.

Altering column types

To alter the type of a column, use the `types=` argument:

```
# Convert the 'age' column to an integer, and 'weight' to a float
table.transform(types={"age": int, "weight": float})
```

See *Adding columns* for a list of available types.

Renaming columns

The `rename=` parameter can rename columns:

```
# Rename 'age' to 'initial_age':
table.transform(rename={"age": "initial_age"})
```

Dropping columns

To drop columns, pass them in the `drop=` set:

```
# Drop the 'age' column:
table.transform(drop={"age"})
```

Changing primary keys

To change the primary key for a table, use `pk=`. This can be passed a single column for a regular primary key, or a tuple of columns to create a compound primary key. Passing `pk=None` will remove the primary key and convert the table into a `rowid` table.

```
# Make `user_id` the new primary key
table.transform(pk="user_id")
```

Changing not null status

You can change the `NOT NULL` status of columns by using `not_null=`. You can pass this a set of columns to make those columns `NOT NULL`:

```
# Make the 'age' and 'weight' columns NOT NULL
table.transform(not_null={"age", "weight"})
```

If you want to take existing `NOT NULL` columns and change them to allow null values, you can do so by passing a dictionary of true/false values instead:

```
# 'age' is NOT NULL but we want to allow NULL:
table.transform(not_null={"age": False})

# Make age allow NULL and switch weight to being NOT NULL:
table.transform(not_null={"age": False, "weight": True})
```

Altering column defaults

The `defaults=` parameter can be used to set or change the defaults for different columns:

```
# Set default age to 1:
table.transform(defaults={"age": 1})

# Now remove the default from that column:
table.transform(defaults={"age": None})
```

Changing column order

The `column_order=` parameter can be used to change the order of the columns. If you pass the names of a subset of the columns those will go first and columns you omitted will appear in their existing order after them.

```
# Change column order
table.transform(column_order=("name", "age", "id")
```

Dropping foreign key constraints

You can use `.transform()` to remove foreign key constraints from a table.

This example drops two foreign keys - the one from `places.country` to `country.id` and the one from `places.continent` to `continent.id`:

```
db["places"].transform(
    drop_foreign_keys=("country", "continent")
)
```

Custom transformations with .transform_sql()

The `.transform()` method can handle most cases, but it does not automatically upgrade indexes, views or triggers associated with the table that is being transformed.

If you want to do something more advanced, you can call the `table.transform_sql(...)` method with the same arguments that you would have passed to `table.transform(...)`.

This method will return a list of SQL statements that should be executed to implement the change. You can then make modifications to that SQL - or add additional SQL statements - before executing it yourself.

1.3.28 Extracting columns into a separate table

The `table.extract()` method can be used to extract specified columns into a separate table.

Imagine a `Trees` table that looks like this:

id	TreeAddress	Species
1	52 Vine St	Palm
2	12 Draft St	Oak
3	51 Dark Ave	Palm
4	1252 Left St	Palm

The `Species` column contains duplicate values. This database could be improved by extracting that column out into a separate `Species` table and pointing to it using a foreign key column.

The schema of the above table is:

```
CREATE TABLE [Trees] (
    [id] INTEGER PRIMARY KEY,
    [TreeAddress] TEXT,
    [Species] TEXT
)
```

Here's how to extract the `Species` column using `.extract()`:

```
db["Trees"].extract("Species")
```

After running this code the table schema now looks like this:

```
CREATE TABLE "Trees" (
    [id] INTEGER PRIMARY KEY,
    [TreeAddress] TEXT,
    [Species_id] INTEGER,
    FOREIGN KEY(Species_id) REFERENCES Species(id)
)
```

A new `Species` table will have been created with the following schema:

```
CREATE TABLE [Species] (
    [id] INTEGER PRIMARY KEY,
    [Species] TEXT
)
```

The `.extract()` method defaults to creating a table with the same name as the column that was extracted, and adding a foreign key column called `tablename_id`.

You can specify a custom table name using `table=`, and a custom foreign key name using `fk_column=`. This example creates a table called `tree_species` and a foreign key column called `tree_species_id`:

```
db["Trees"].extract("Species", table="tree_species", fk_column="tree_species_id")
```

The resulting schema looks like this:

```
CREATE TABLE "Trees" (
    [id] INTEGER PRIMARY KEY,
    [TreeAddress] TEXT,
    [tree_species_id] INTEGER,
    FOREIGN KEY(tree_species_id) REFERENCES tree_species(id)
)

CREATE TABLE [tree_species] (
    [id] INTEGER PRIMARY KEY,
    [Species] TEXT
)
```

You can also extract multiple columns into the same external table. Say for example you have a table like this:

id	TreeAddress	CommonName	LatinName
1	52 Vine St	Palm	Arecaceae
2	12 Draft St	Oak	Quercus
3	51 Dark Ave	Palm	Arecaceae
4	1252 Left St	Palm	Arecaceae

You can pass `["CommonName", "LatinName"]` to `.extract()` to extract both of those columns:

```
db["Trees"].extract(["CommonName", "LatinName"])
```

This produces the following schema:

```
CREATE TABLE "Trees" (
    [id] INTEGER PRIMARY KEY,
    [TreeAddress] TEXT,
    [CommonName_LatinName_id] INTEGER,
    FOREIGN KEY(CommonName_LatinName_id) REFERENCES CommonName_LatinName(id)
)
CREATE TABLE [CommonName_LatinName] (
    [id] INTEGER PRIMARY KEY,
    [CommonName] TEXT,
    [LatinName] TEXT
)
```

The table name `CommonName_LatinName` is derived from the extract columns. You can use `table=` and `fk_column=` to specify custom names like this:

```
db["Trees"].extract(["CommonName", "LatinName"], table="Species", fk_column="species_
↪id")
```

This produces the following schema:

```
CREATE TABLE "Trees" (
    [id] INTEGER PRIMARY KEY,
    [TreeAddress] TEXT,
    [species_id] INTEGER,
    FOREIGN KEY(species_id) REFERENCES Species(id)
)
CREATE TABLE [Species] (
    [id] INTEGER PRIMARY KEY,
    [CommonName] TEXT,
    [LatinName] TEXT
)
```

You can use the `rename=` argument to rename columns in the lookup table. To create a `Species` table with columns called `name` and `latin` you can do this:

```
db["Trees"].extract(
    ["CommonName", "LatinName"],
    table="Species",
    fk_column="species_id",
    rename={"CommonName": "name", "LatinName": "latin"}
)
```

This produces a lookup table like so:

```
CREATE TABLE [Species] (
    [id] INTEGER PRIMARY KEY,
    [name] TEXT,
    [latin] TEXT
)
```

1.3.29 Setting an ID based on the hash of the row contents

Sometimes you will find yourself working with a dataset that includes rows that do not have a provided obvious ID, but where you would like to assign one so that you can later upsert into that table without creating duplicate records.

In these cases, a useful technique is to create an ID that is derived from the sha1 hash of the row contents.

`sqlite-utils` can do this for you using the `hash_id=` option. For example:

```
db = sqlite_utils.Database("dogs.db")
db["dogs"].upsert({"name": "Cleo", "twitter": "cleopaws"}, hash_id="id")
print(list(db["dogs"]))
```

Outputs:

```
[{'id': 'f501265970505d9825d8d9f590bfab3519fb20b1', 'name': 'Cleo', 'twitter':
↪'cleopaws'}]
```

If you are going to use that ID straight away, you can access it using `last_pk`:

```
dog_id = db["dogs"].upsert({
    "name": "Cleo",
    "twitter": "cleopaws"
}, hash_id="id").last_pk
# dog_id is now "f501265970505d9825d8d9f590bfab3519fb20b1"
```

1.3.30 Creating views

The `.create_view()` method on the database class can be used to create a view:

```
db.create_view("good_dogs", """
    select * from dogs where is_good_dog = 1
""")
```

This will raise a `sqlite_utils.utils.OperationalError` if a view with that name already exists.

You can pass `ignore=True` to silently ignore an existing view and do nothing, or `replace=True` to replace an existing view with a new definition if your select statement differs from the current view:

```
db.create_view("good_dogs", """
    select * from dogs where is_good_dog = 1
""", replace=True)
```

1.3.31 Storing JSON

SQLite has excellent JSON support, and `sqlite-utils` can help you take advantage of this: if you attempt to insert a value that can be represented as a JSON list or dictionary, `sqlite-utils` will create TEXT column and store your data as serialized JSON. This means you can quickly store even complex data structures in SQLite and query them using JSON features.

For example:

```
db["niche_museums"].insert({
    "name": "The Bigfoot Discovery Museum",
    "url": "http://bigfootdiscoveryproject.com/"
    "hours": {
        "Monday": [11, 18],
        "Wednesday": [11, 18],
        "Thursday": [11, 18],
        "Friday": [11, 18],
        "Saturday": [11, 18],
        "Sunday": [11, 18]
    },
    "address": {
        "streetAddress": "5497 Highway 9",
        "addressLocality": "Felton, CA",
        "postalCode": "95018"
    }
})
db.execute("""
    select json_extract(address, '$.addressLocality')
    from niche_museums
""").fetchall()
# Returns [('Felton, CA',)]
```

1.3.32 Converting column values using SQL functions

Sometimes it can be useful to run values through a SQL function prior to inserting them. A simple example might be converting a value to upper case while it is being inserted.

The `conversions={...}` parameter can be used to specify custom SQL to be used as part of a `INSERT` or `UPDATE` SQL statement.

You can specify an upper case conversion for a specific column like so:

```
db["example"].insert({
    "name": "The Bigfoot Discovery Museum"
}, conversions={"name": "upper(?)"})

# list(db["example"].rows) now returns:
# [{'name': 'THE BIGFOOT DISCOVERY MUSEUM'}]
```

The dictionary key is the column name to be converted. The value is the SQL fragment to use, with a `?` placeholder for the original value.

A more useful example: if you are working with SpatiaLite you may find yourself wanting to create geometry values from a WKT value. Code to do that could look like this:

```
import sqlite3
import sqlite_utils
from shapely.geometry import shape
import requests

# Open a database and load the SpatiaLite extension:
import sqlite3

conn = sqlite3.connect("places.db")
conn.enable_load_extension(True)
conn.load_extension("/usr/local/lib/mod_spatialite.dylib")

# Use sqlite-utils to create a places table:
db = sqlite_utils.Database(conn)
places = db["places"].create({"id": int, "name": str,})

# Add a SpatiaLite 'geometry' column:
db.execute("select InitSpatialMetadata(1)")
db.execute(
    "SELECT AddGeometryColumn('places', 'geometry', 4326, 'MULTIPOLYGON', 2);"
)

# Fetch some GeoJSON from Who's On First:
geojson = requests.get(
    "https://data.whosonfirst.org/404/227/475/404227475.geojson"
).json()

# Convert to "Well Known Text" format using shapely
wkt = shape(geojson["geometry"]).wkt

# Insert the record, converting the WKT to a SpatiaLite geometry:
db["places"].insert(
    {"name": "Wales", "geometry": wkt},
    conversions={"geometry": "GeomFromText(?, 4326)"},
)
```

1.3.33 Introspection

If you have loaded an existing table or view, you can use introspection to find out more about it:

```
>>> db["PlantType"]
<Table PlantType (id, value)>
```

.exists()

The `.exists()` method can be used to find out if a table exists or not:

```
>>> db["PlantType"].exists()
True
>>> db["PlantType2"].exists()
False
```

.count

The `.count` property shows the current number of rows (`select count(*) from table`):

```
>>> db["PlantType"].count
3
>>> db["Street_Tree_List"].count
189144
```

This property will take advantage of *Cached table counts using triggers* if the `use_counts_table` property is set on the database. You can avoid that optimization entirely by calling `table.count_where()` instead of accessing the property.

.columns

The `.columns` property shows the columns in the table or view. It returns a list of `Column(cid, name, type, notnull, default_value, is_pk)` named tuples.

```
>>> db["PlantType"].columns
[Column(cid=0, name='id', type='INTEGER', notnull=0, default_value=None, is_pk=1),
 Column(cid=1, name='value', type='TEXT', notnull=0, default_value=None, is_pk=0)]
```

.columns_dict

The `.columns_dict` property returns a dictionary version of the columns with just the names and Python types:

```
>>> db["PlantType"].columns_dict
{'id': <class 'int'>, 'value': <class 'str'>}
```

.pks

The `.pks` property returns a list of strings naming the primary key columns for the table:

```
>>> db["PlantType"].pks
['id']
```

If a table has no primary keys but is a rowid table, this property will return `['rowid']`.

.use_rowid

Almost all SQLite tables have a `rowid` column, but a table with no explicitly defined primary keys must use that `rowid` as the primary key for identifying individual rows. The `.use_rowid` property checks to see if a table needs to use the `rowid` in this way - it returns `True` if the table has no explicitly defined primary keys and `False` otherwise.

```
>>> db["PlantType"].use_rowid
False
```

.foreign_keys

The `.foreign_keys` property returns any foreign key relationships for the table, as a list of `ForeignKey(table, column, other_table, other_column)` named tuples. It is not available on views.

```
>>> db["Street_Tree_List"].foreign_keys
[ForeignKey(table='Street_Tree_List', column='qLegalStatus', other_table='qLegalStatus
↪', other_column='id'),
 ForeignKey(table='Street_Tree_List', column='qCareAssistant', other_table=
↪'qCareAssistant', other_column='id'),
 ForeignKey(table='Street_Tree_List', column='qSiteInfo', other_table='qSiteInfo',␣
↪other_column='id'),
 ForeignKey(table='Street_Tree_List', column='qSpecies', other_table='qSpecies',␣
↪other_column='id'),
 ForeignKey(table='Street_Tree_List', column='qCaretaker', other_table='qCaretaker',␣
↪other_column='id'),
 ForeignKey(table='Street_Tree_List', column='PlantType', other_table='PlantType',␣
↪other_column='id')]
```

.schema

The `.schema` property outputs the table's schema as a SQL string:

```
>>> print(db["Street_Tree_List"].schema)
CREATE TABLE "Street_Tree_List" (
"TreeID" INTEGER,
  "qLegalStatus" INTEGER,
  "qSpecies" INTEGER,
  "qAddress" TEXT,
  "SiteOrder" INTEGER,
  "qSiteInfo" INTEGER,
  "PlantType" INTEGER,
  "qCaretaker" INTEGER,
  "qCareAssistant" INTEGER,
  "PlantDate" TEXT,
  "DBH" INTEGER,
  "PlotSize" TEXT,
  "PermitNotes" TEXT,
  "XCoord" REAL,
  "YCoord" REAL,
  "Latitude" REAL,
```

(continues on next page)

(continued from previous page)

```
  "Longitude" REAL,
  "Location" TEXT
,
FOREIGN KEY ("PlantType") REFERENCES [PlantType](id),
    FOREIGN KEY ("qCaretaker") REFERENCES [qCaretaker](id),
    FOREIGN KEY ("qSpecies") REFERENCES [qSpecies](id),
    FOREIGN KEY ("qSiteInfo") REFERENCES [qSiteInfo](id),
    FOREIGN KEY ("qCareAssistant") REFERENCES [qCareAssistant](id),
    FOREIGN KEY ("qLegalStatus") REFERENCES [qLegalStatus](id))
```

.indexes

The `.indexes` property returns all indexes created for a table, as a list of `Index(seq, name, unique, origin, partial, columns)` named tuples. It is not available on views.

```
>>> db["Street_Tree_List"].indexes
[Index(seq=0, name='"Street_Tree_List_qLegalStatus"', unique=0, origin='c', partial=0,
↪ columns=['qLegalStatus']),
 Index(seq=1, name='"Street_Tree_List_qCareAssistant"', unique=0, origin='c',␣
↪partial=0, columns=['qCareAssistant']),
 Index(seq=2, name='"Street_Tree_List_qSiteInfo"', unique=0, origin='c', partial=0,␣
↪columns=['qSiteInfo']),
 Index(seq=3, name='"Street_Tree_List_qSpecies"', unique=0, origin='c', partial=0,␣
↪columns=['qSpecies']),
 Index(seq=4, name='"Street_Tree_List_qCaretaker"', unique=0, origin='c', partial=0,␣
↪columns=['qCaretaker']),
 Index(seq=5, name='"Street_Tree_List_PlantType"', unique=0, origin='c', partial=0,␣
↪columns=['PlantType'])]
```

.xindexes

The `.xindexes` property returns more detailed information about the indexes on the table, using the SQLite PRAGMA index_xinfo() mechanism. It returns a list of `XIndex(name, columns)` named tuples, where `columns` is a list of `XIndexColumn(seqno, cid, name, desc, coll, key)` named tuples.

```
>>> db["ny_times_us_counties"].xindexes
[
    XIndex(
        name='idx_ny_times_us_counties_date',
        columns=[
            XIndexColumn(seqno=0, cid=0, name='date', desc=1, coll='BINARY', key=1),
            XIndexColumn(seqno=1, cid=-1, name=None, desc=0, coll='BINARY', key=0)
        ]
    ),
    XIndex(
        name='idx_ny_times_us_counties_fips',
        columns=[
            XIndexColumn(seqno=0, cid=3, name='fips', desc=0, coll='BINARY', key=1),
            XIndexColumn(seqno=1, cid=-1, name=None, desc=0, coll='BINARY', key=0)
        ]
    )
]
```

.triggers

The `.triggers` property lists database triggers. It can be used on both database and table objects. It returns a list of `Trigger(name, table, sql)` named tuples.

```
>>> db["authors"].triggers
[Trigger(name='authors_ai', table='authors', sql='CREATE TRIGGER [authors_ai] AFTER␣
↪INSERT...'),
 Trigger(name='authors_ad', table='authors', sql="CREATE TRIGGER [authors_ad] AFTER␣
↪DELETE..."),
 Trigger(name='authors_au', table='authors', sql="CREATE TRIGGER [authors_au] AFTER␣
↪UPDATE")]
>>> db.triggers
... similar output to db["authors"].triggers
```

.triggers_dict

The `.triggers_dict` property returns the triggers for that table as a dictionary mapping their names to their SQL definitions.

```
>>> db["authors"].triggers_dict
{'authors_ai': 'CREATE TRIGGER [authors_ai] AFTER INSERT...',
 'authors_ad': 'CREATE TRIGGER [authors_ad] AFTER DELETE...',
 'authors_au': 'CREATE TRIGGER [authors_au] AFTER UPDATE'}
```

The same property exists on the database, and will return all triggers across all tables:

```
>>> db.triggers_dict
{'authors_ai': 'CREATE TRIGGER [authors_ai] AFTER INSERT...',
 'authors_ad': 'CREATE TRIGGER [authors_ad] AFTER DELETE...',
 'authors_au': 'CREATE TRIGGER [authors_au] AFTER UPDATE'}
```

.detect_fts()

The `detect_fts()` method returns the associated SQLite FTS table name, if one exists for this table. If the table has not been configured for full-text search it returns `None`.

```
>>> db["authors"].detect_fts()
"authors_fts"
```

.virtual_table_using

The `.virtual_table_using` property reveals if a table is a virtual table. It returns `None` for regular tables and the upper case version of the type of virtual table otherwise. For example:

```
>>> db["authors"].enable_fts(["name"])
>>> db["authors_fts"].virtual_table_using
"FTS5"
```

.has_counts_triggers

The `.has_counts_triggers` property shows if a table has been configured with triggers for updating a `_counts` table, as described in *Cached table counts using triggers*.

```
>>> db["authors"].has_counts_triggers
False
>>> db["authors"].enable_counts()
>>> db["authors"].has_counts_triggers
True
```

1.3.34 Enabling full-text search

You can enable full-text search on a table using `.enable_fts(columns)`:

```
db["dogs"].enable_fts(["name", "twitter"])
```

You can then run searches using the `.search()` method:

```
rows = list(db["dogs"].search("cleo"))
```

This method returns a generator that can be looped over to get dictionaries for each row, similar to *Listing rows*.

If you insert additional records into the table you will need to refresh the search index using `populate_fts()`:

```
db["dogs"].insert({
    "id": 2,
    "name": "Marnie",
    "twitter": "MarnieTheDog",
    "age": 16,
    "is_good_dog": True,
}, pk="id")
db["dogs"].populate_fts(["name", "twitter"])
```

A better solution is to use database triggers. You can set up database triggers to automatically update the full-text index using `create_triggers=True`:

```
db["dogs"].enable_fts(["name", "twitter"], create_triggers=True)
```

`.enable_fts()` defaults to using FTS5. If you wish to use FTS4 instead, use the following:

```
db["dogs"].enable_fts(["name", "twitter"], fts_version="FTS4")
```

You can customize the tokenizer configured for the table using the `tokenize=` parameter. For example, to enable Porter stemming, where English words like "running" will match stemmed alternatives such as "run", use `tokenize="porter"`:

```
db["articles"].enable_fts(["headline", "body"], tokenize="porter")
```

The SQLite documentation has more on FTS5 tokenizers and FTS4 tokenizers. `porter` is a valid option for both.

If you attempt to configure a FTS table where one already exists, a `sqlite3.OperationalError` exception will be raised.

You can replace the existing table with a new configuration using `replace=True`:

```
db["articles"].enable_fts(["headline"], tokenize="porter", replace=True)
```

This will have no effect if the FTS table already exists, otherwise it will drop and recreate the table with the new settings. This takes into consideration the columns, the tokenizer, the FTS version used and whether or not the table has triggers.

To remove the FTS tables and triggers you created, use the `disable_fts()` table method:

```
db["dogs"].disable_fts()
```

Searching with table.search()

The `table.search(q)` method returns a generator over Python dictionaries representing rows that match the search phrase `q`, ordered by relevance with the most relevant results first.

```
for article in db["articles"].search("jquery"):
    print(article)
```

The `.search()` method also accepts the following optional parameters:

`order_by` string The column to sort by. Defaults to relevance score. Can optionally include a `desc`, e.g. `rowid desc`.

`columns` array of strings Columns to return. Defaults to all columns.

`limit` integer Number of results to return. Defaults to all results.

`offset` integer Offset to use along side the limit parameter.

To return just the title and published columns for three matches for `"dog"` ordered by `published` with the most recent first, use the following:

```
for article in db["articles"].search(
    "dog",
    order_by="published desc",
    limit=3,
    columns=["title", "published"]
):
    print(article)
```

Building SQL queries with table.search_sql()

You can generate the SQL query that would be used for a search using the `table.search_sql()` method. It takes the same arguments as `table.search()` with the exception of the search query itself, since the returned SQL includes a parameter that can be used for the search.

```
print(db["articles"].search_sql(columns=["title", "author"]))
```

Outputs:

```
with original as (
    select
        rowid,
        [title],
        [author]
    from [articles]
```

(continues on next page)

(continued from previous page)

```
)
select
    [original].[title],
    [original].[author]
from
    [original]
    join [articles_fts] on [original].rowid = [articles_fts].rowid
where
    [articles_fts] match :query
order by
    [articles_fts].rank
```

This method detects if a SQLite table uses FTS4 or FTS5, and outputs the correct SQL for ordering by relevance depending on the search type.

The FTS4 output looks something like this:

```
with original as (
    select
        rowid,
        [title],
        [author]
    from [articles]
)
select
    [original].[title],
    [original].[author]
from
    [original]
    join [articles_fts] on [original].rowid = [articles_fts].rowid
where
    [articles_fts] match :query
order by
    rank_bm25(matchinfo([articles_fts], 'pcnalx'))
```

This uses the `rank_bm25()` custom SQL function from sqlite-fts4. You can register that custom function against a `Database` connection using this method:

```
db.register_fts4_bm25()
```

1.3.35 Rebuilding a full-text search table

You can rebuild a table using the `table.rebuild_fts()` method. This is useful for if the table configuration changes or the indexed data has become corrupted in some way.

```
db["dogs"].rebuild_fts()
```

This method can be called on a table that has been configured for full-text search - `dogs` in this instance - or directly on a `_fts` table:

```
db["dogs_fts"].rebuild_fts()
```

This runs the following SQL:

```
INSERT INTO dogs_fts (dogs_fts) VALUES ("rebuild");
```

1.3.36 Optimizing a full-text search table

Once you have populated a FTS table you can optimize it to dramatically reduce its size like so:

```
db["dogs"].optimize()
```

This runs the following SQL:

```
INSERT INTO dogs_fts (dogs_fts) VALUES ("optimize");
```

1.3.37 Cached table counts using triggers

The `select count(*)` query in SQLite requires a full scan of the primary key index, and can take an increasingly long time as the table grows larger.

The `table.enable_counts()` method can be used to configure triggers to continuously update a record in a `_counts` table. This value can then be used to quickly retrieve the count of rows in the associated table.

```
db["dogs"].enable_counts()
```

This will create the `_counts` table if it does not already exist, with the following schema:

```
CREATE TABLE [_counts] (
   [table] TEXT PRIMARY KEY,
   [count] INTEGER DEFAULT 0
)
```

You can enable cached counts for every table in a database (except for virtual tables and the `_counts` table itself) using the database `enable_counts()` method:

```
db.enable_counts()
```

Once enabled, table counts will be stored in the `_counts` table. The count records will be automatically kept up-to-date by the triggers when rows are added or deleted to the table.

To access these counts you can query the `_counts` table directly or you can use the `db.cached_counts()` method. This method returns a dictionary mapping tables to their counts:

```
>>> db.cached_counts()
{'global-power-plants': 33643,
 'global-power-plants_fts_data': 136,
 'global-power-plants_fts_idx': 199,
 'global-power-plants_fts_docsize': 33643,
 'global-power-plants_fts_config': 1}
```

You can pass a list of table names to this method to retrieve just those counts:

```
>>> db.cached_counts(["global-power-plants"])
{'global-power-plants': 33643}
```

The `table.count` property executes a `select count(*)` query by default, unless the `db.use_counts_table` property is set to `True`.

You can set `use_counts_table` to `True` when you instantiate the database object:

```
db = Database("global-power-plants.db", use_counts_table=True)
```

If the property is `True` any calls to the `table.count` property will first attempt to find the cached count in the `_counts` table, and fall back on a `count(*)` query if the value is not available or the table is missing.

Calling the `.enable_counts()` method on a database or table object will set `use_counts_table` to `True` for the lifetime of that database object.

If the `_counts` table ever becomes out-of-sync with the actual table counts you can repair it using the `.reset_counts()` method:

```
db.reset_counts()
```

1.3.38 Creating indexes

You can create an index on a table using the `.create_index(columns)` method. The method takes a list of columns:

```
db["dogs"].create_index(["is_good_dog"])
```

By default the index will be named `idx_{table-name}_{columns}` - if you want to customize the name of the created index you can pass the `index_name` parameter:

```
db["dogs"].create_index(
    ["is_good_dog", "age"],
    index_name="good_dogs_by_age"
)
```

To create an index in descending order for a column, wrap the column name in `db.DescIndex()` like this:

```
from sqlite_utils.db import DescIndex

db["dogs"].create_index(
    ["is_good_dog", DescIndex("age")],
    index_name="good_dogs_by_age"
)
```

You can create a unique index by passing `unique=True`:

```
db["dogs"].create_index(["name"], unique=True)
```

Use `if_not_exists=True` to do nothing if an index with that name already exists.

1.3.39 Vacuum

You can optimize your database by running VACUUM against it like so:

```
Database("my_database.db").vacuum()
```

1.3.40 WAL mode

You can enable Write-Ahead Logging for a database with `.enable_wal()`:

```
Database("my_database.db").enable_wal()
```

You can disable WAL mode using `.disable_wal()`:

```
Database("my_database.db").disable_wal()
```

You can check the current journal mode for a database using the `journal_mode` property:

```
journal_mode = Database("my_database.db").journal_mode
```

This will usually be `wal` or `delete` (meaning WAL is disabled), but can have other values - see the PRAGMA journal_mode documentation.

1.3.41 Suggesting column types

When you create a new table for a list of inserted or upserted Python dictionaries, those methods detect the correct types for the database columns based on the data you pass in.

In some situations you may need to intervene in this process, to customize the columns that are being created in some way - see *Explicitly creating a table*.

That table `.create()` method takes a dictionary mapping column names to the Python type they should store:

```
db["cats"].create({
    "id": int,
    "name": str,
    "weight": float,
})
```

You can use the `suggest_column_types()` helper function to derive a dictionary of column names and types from a list of records, suitable to be passed to `table.create()`.

For example:

```
from sqlite_utils import Database, suggest_column_types

cats = [{
    "id": 1,
    "name": "Snowflake"
}, {
    "id": 2,
    "name": "Crabtree",
    "age": 4
}]
types = suggest_column_types(cats)
# types now looks like this:
# {"id": <class 'int'>,
#  "name": <class 'str'>,
#  "age": <class 'int'>}

# Manually add an extra field:
types["thumbnail"] = bytes
# types now looks like this:
# {"id": <class 'int'>,
#  "name": <class 'str'>,
#  "age": <class 'int'>,
#  "thumbnail": <class 'bytes'>}
```

(continues on next page)

(continued from previous page)

```
# Create the table
db = Database("cats.db")
db["cats"].create(types, pk="id")
# Insert the records
db["cats"].insert_all(cats)

# list(db["cats"].rows) now returns:
# [{"id": 1, "name": "Snowflake", "age": None, "thumbnail": None}
#  {"id": 2, "name": "Crabtree", "age": 4, "thumbnail": None}]

# The table schema looks like this:
# print(db["cats"].schema)
# CREATE TABLE [cats] (
#    [id] INTEGER PRIMARY KEY,
#    [name] TEXT,
#    [age] INTEGER,
#    [thumbnail] BLOB
# )
```

1.3.42 Finding SpatiaLite

The `find_spatialite()` function searches for the SpatiaLite SQLite extension in some common places. It returns a string path to the location, or `None` if SpatiaLite was not found.

You can use it in code like this:

```
from sqlite_utils import Database
from sqlite_utils.utils import find_spatialite

db = Database("mydb.db")
spatialite = find_spatialite()
if spatialite:
    db.conn.enable_load_extension(True)
    db.conn.load_extension(spatialite)
```

1.3.43 Registering custom SQL functions

SQLite supports registering custom SQL functions written in Python. The `db.register_function()` method lets you register these functions, and keeps track of functions that have already been registered.

If you use it as a method it will automatically detect the name and number of arguments needed by the function:

```
from sqlite_utils import Database

db = Database(memory=True)

def reverse_string(s):
    return "".join(reversed(list(s)))

db.register_function(reverse_string)
print(db.execute('select reverse_string("hello")').fetchone()[0])
# This prints "olleh"
```

You can also use the method as a function decorator like so:

```
@db.register_function
def reverse_string(s):
    return "".join(reversed(list(s)))

print(db.execute('select reverse_string("hello")').fetchone()[0])
```

Python 3.8 added the ability to register deterministic SQLite functions, allowing you to indicate that a function will return the exact same result for any given inputs and hence allowing SQLite to apply some performance optimizations. You can mark a function as deterministic using `deterministic=True`, like this:

```
@db.register_function(deterministic=True)
def reverse_string(s):
    return "".join(reversed(list(s)))
```

If you run this on a version of Python prior to 3.8 your code will still work, but the `deterministic=True` parameter will be ignored.

By default registering a function with the same name and number of arguments will have no effect - the `Database` instance keeps track of functions that have already been registered and skips registering them if `@db.register_function` is called a second time.

If you want to deliberately replace the registered function with a new implementation, use the `replace=True` argument:

```
@db.register_function(deterministic=True, replace=True)
def reverse_string(s):
    return s[::-1]
```

Exceptions that occur inside a user-defined function default to returning the following error:

```
Unexpected error: user-defined function raised exception
```

You can cause `sqlite3` to return more useful errors, including the traceback from the custom function, by executing the following before your custom functions are executed:

```
from sqlite_utils.utils import sqlite3

sqlite3.enable_callback_tracebacks(True)
```

1.3.44 Quoting strings for use in SQL

In almost all cases you should pass values to your SQL queries using the optional `parameters` argument to `db.query()`, as described in *Passing parameters*.

If that option isn't relevant to your use-case you can to quote a string for use with SQLite using the `db.quote()` method, like so:

```
>>> db = Database(memory=True)
>>> db.quote("hello")
"'hello'"
>>> db.quote("hello'this'has'quotes")
"'hello''this''has''quotes'"
```

1.4 Contributing

To work on this library locally, first checkout the code. Then create a new virtual environment:

```
git clone git@github.com:simonw/sqlite-utils
cd sqlite-utils
python3 -mvenv venv
source venv/bin/activate
```

Or if you are using `pipenv`:

```
pipenv shell
```

Within the virtual environment running `sqlite-utils` should run your locally editable version of the tool. You can use `which sqlite-utils` to confirm that you are running the version that lives in your virtual environment.

1.4.1 Running the tests

To install the dependencies and test dependencies:

```
pip install -e '.[test]'
```

To run the tests:

```
pytest
```

1.4.2 Building the documentation

To build the documentation, first install the documentation dependencies:

```
pip install -e '.[docs]'
```

Then run `make livehtml` from the `docs/` directory to start a server on port 8000 that will serve the documentation and live-reload any time you make an edit to a `.rst` file:

```
cd docs
make livehtml
```

1.4.3 Linting and formatting

`sqlite-utils` uses Black for code formatting, and flake8 and mypy for linting and type checking.

Black is installed as part of `pip install -e '.[test]'` - you can then format your code by running it in the root of the project:

```
black .
```

To install `mypy` and `flake8` run the following:

```
pip install -e '.[flake8,mypy]'
```

Both commands can then be run in the root of the project like this:

```
flake8
mypy sqlite_utils
```

All three of these tools are run by our CI mechanism against every commit and pull request.

1.5 Changelog

1.5.1 3.15 (2021-08-09)

- `sqlite-utils insert --flatten` option for *flattening nested JSON objects* to create tables with column names like `topkey_nestedkey`. (#310)
- Fixed several spelling mistakes in the documentation, spotted using codespell.
- Errors that occur while using the `sqlite-utils` CLI tool now show the responsible SQL and query parameters, if possible. (#309)

1.5.2 3.14 (2021-08-02)

This release introduces the new *sqlite-utils convert command* (#251) and corresponding *table.convert(...)* Python method (#302). These tools can be used to apply a Python conversion function to one or more columns of a table, either updating the column in place or using transformed data from that column to populate one or more other columns.

This command-line example uses the Python standard library textwrap module to wrap the content of the `content` column in the `articles` table to 100 characters:

```
$ sqlite-utils convert content.db articles content \
    '"\n".join(textwrap.wrap(value, 100))' \
    --import=textwrap
```

The same operation in Python code looks like this:

```
import sqlite_utils, textwrap

db = sqlite_utils.Database("content.db")
db["articles"].convert("content", lambda v: "\n".join(textwrap.wrap(v, 100)))
```

See the full documentation for the *sqlite-utils convert command* and the *table.convert(...)* Python method for more details.

Also in this release:

- The new `table.count_where(...)` method, for counting rows in a table that match a specific SQL `WHERE` clause. (#305)
- New `--silent` option for the *sqlite-utils insert-files command* to hide the terminal progress bar, consistent with the `--silent` option for `sqlite-utils convert`. (#301)

1.5.3 3.13 (2021-07-24)

- `sqlite-utils schema my.db table1 table2` command now accepts optional table names. (#299)
- `sqlite-utils memory --help` now describes the `--schema` option.

1.5.4 3.12 (2021-06-25)

- New *db.query(sql, params)* method, which executes a SQL query and returns the results as an iterator over Python dictionaries. (#290)
- This project now uses `flake8` and has started to use `mypy`. (#291)
- New documentation on *contributing* to this project. (#292)

1.5.5 3.11 (2021-06-20)

- New `sqlite-utils memory data.csv --schema` option, for outputting the schema of the in-memory database generated from one or more files. See *–schema, –dump and –save*. (#288)
- Added *installation instructions*. (#286)

1.5.6 3.10 (2021-06-19)

This release introduces the `sqlite-utils memory` command, which can be used to load CSV or JSON data into a temporary in-memory database and run SQL queries (including joins across multiple files) directly against that data.

Also new: `sqlite-utils insert --detect-types`, `sqlite-utils dump`, `table.use_rowid` plus some smaller fixes.

sqlite-utils memory

This example of `sqlite-utils memory` retrieves information about the all of the repositories in the Dogsheep organization on GitHub using this JSON API, sorts them by their number of stars and outputs a table of the top five (using `-t`):

```
$ curl -s 'https://api.github.com/users/dogsheep/repos' \
  | sqlite-utils memory - '
      select full_name, forks_count, stargazers_count
      from stdin order by stargazers_count desc limit 5
    ' -t
full_name                          forks_count    stargazers_count
---------------------------------  -------------  ------------------
dogsheep/twitter-to-sqlite                  12                 225
dogsheep/github-to-sqlite                   14                 139
dogsheep/dogsheep-photos                     5                 116
dogsheep/dogsheep.github.io                  7                  90
dogsheep/healthkit-to-sqlite                 4                  85
```

The tool works against files on disk as well. This example joins data from two CSV files:

```
$ cat creatures.csv
species_id,name
1,Cleo
2,Bants
2,Dori
2,Azi
$ cat species.csv
id,species_name
1,Dog
2,Chicken
```

(continues on next page)

(continued from previous page)

```
$ sqlite-utils memory species.csv creatures.csv '
  select * from creatures join species on creatures.species_id = species.id
'
[{"species_id": 1, "name": "Cleo", "id": 1, "species_name": "Dog"},
 {"species_id": 2, "name": "Bants", "id": 2, "species_name": "Chicken"},
 {"species_id": 2, "name": "Dori", "id": 2, "species_name": "Chicken"},
 {"species_id": 2, "name": "Azi", "id": 2, "species_name": "Chicken"}]
```

Here the `species.csv` file becomes the `species` table, the `creatures.csv` file becomes the `creatures` table and the output is JSON, the default output format.

You can also use the `--attach` option to attach existing SQLite database files to the in-memory database, in order to join data from CSV or JSON directly against your existing tables.

Full documentation of this new feature is available in *Querying data directly using an in-memory database*. (#272)

sqlite-utils insert –detect-types

The *sqlite-utils insert* command can be used to insert data from JSON, CSV or TSV files into a SQLite database file. The new `--detect-types` option (shortcut `-d`), when used in conjunction with a CSV or TSV import, will automatically detect if columns in the file are integers or floating point numbers as opposed to treating everything as a text column and create the new table with the corresponding schema. See *Inserting CSV or TSV data* for details. (#282)

Other changes

- **Bug fix**: `table.transform()`, when run against a table without explicit primary keys, would incorrectly create a new version of the table with an explicit primary key column called `rowid`. (#284)
- New `table.use_rowid` introspection property, see *.use_rowid*. (#285)
- The new `sqlite-utils dump file.db` command outputs a SQL dump that can be used to recreate a database. (#274)
- `-h` now works as a shortcut for `--help`, thanks Loren McIntyre. (#276)
- Now using pytest-cov and Codecov to track test coverage - currently at 96%. (#275)
- SQL errors that occur when using `sqlite-utils query` are now displayed as CLI errors.

1.5.7 3.9.1 (2021-06-12)

- Fixed bug when using `table.upsert_all()` to create a table with only a single column that is treated as the primary key. (#271)

1.5.8 3.9 (2021-06-11)

- New `sqlite-utils schema` command showing the full SQL schema for a database, see *Showing the schema (CLI)*. (#268)
- `db.schema` introspection property exposing the same feature to the Python library, see *Showing the schema (Python library)*.

1.5.9 3.8 (2021-06-02)

- New `sqlite-utils indexes` command to list indexes in a database, see *Listing indexes*. (#263)
- `table.xindexes` introspection property returning more details about that table's indexes, see *.xindexes*. (#261)

1.5.10 3.7 (2021-05-28)

- New `table.pks_and_rows_where()` method returning `(primary_key, row_dictionary)` tuples - see *Listing rows with their primary keys*. (#240)
- Fixed bug with `table.add_foreign_key()` against columns containing spaces. (#238)
- `table_or_view.drop(ignore=True)` option for avoiding errors if the table or view does not exist. (#237)
- `sqlite-utils drop-view --ignore` and `sqlite-utils drop-table --ignore` options. (#237)
- Fixed a bug with inserts of nested JSON containing non-ascii strings - thanks, Dylan Wu. (#257)
- Suggest `--alter` if an error occurs caused by a missing column. (#259)
- Support creating indexes with columns in descending order, see *API documentation* and *CLI documentation*. (#260)
- Correctly handle CSV files that start with a UTF-8 BOM. (#250)

1.5.11 3.6 (2021-02-18)

This release adds the ability to execute queries joining data from more than one database file - similar to the cross database querying feature introduced in Datasette 0.55.

- The `db.attach(alias, filepath)` Python method can be used to attach extra databases to the same connection, see *db.attach() in the Python API documentation*. (#113)
- The `--attach` option attaches extra aliased databases to run SQL queries against directly on the command-line, see *attaching additional databases in the CLI documentation*. (#236)

1.5.12 3.5 (2021-02-14)

- `sqlite-utils insert --sniff` option for detecting the delimiter and quote character used by a CSV file, see *Alternative delimiters and quote characters*. (#230)
- The `table.rows_where()`, `table.search()` and `table.search_sql()` methods all now take optional `offset=` and `limit=` arguments. (#231)
- New `--no-headers` option for `sqlite-utils insert --csv` to handle CSV files that are missing the header row, see *CSV files without a header row*. (#228)
- Fixed bug where inserting data with extra columns in subsequent chunks would throw an error. Thanks @nieuwenhoven for the fix. (#234)
- Fixed bug importing CSV files with columns containing more than 128KB of data. (#229)
- Test suite now runs in CI against Ubuntu, macOS and Windows. Thanks @nieuwenhoven for the Windows test fixes. (#232)

1.5.13 3.4.1 (2021-02-05)

- Fixed a code import bug that slipped in to 3.4. (#226)

1.5.14 3.4 (2021-02-05)

- `sqlite-utils insert --csv` now accepts optional `--delimiter` and `--quotechar` options. See *Alternative delimiters and quote characters*. (#223)

1.5.15 3.3 (2021-01-17)

- The `table.m2m()` method now accepts an optional `alter=True` argument to specify that any missing columns should be added to the referenced table. See *Working with many-to-many relationships*. (#222)

1.5.16 3.2.1 (2021-01-12)

- Fixed a bug where `.add_missing_columns()` failed to take case insensitive column names into account. (#221)

1.5.17 3.2 (2021-01-03)

This release introduces a new mechanism for speeding up `count(*)` queries using cached table counts, stored in a `_counts` table and updated by triggers. This mechanism is described in *Cached table counts using triggers*, and can be enabled using Python API methods or the new `enable-counts` CLI command. (#212)

- `table.enable_counts()` method for enabling these triggers on a specific table.
- `db.enable_counts()` method for enabling triggers on every table in the database. (#213)
- New `sqlite-utils enable-counts my.db` command for enabling counts on all or specific tables, see *Enabling cached counts*. (#214)
- New `sqlite-utils triggers` command for listing the triggers defined for a database or specific tables, see *Listing triggers*. (#218)
- New `db.use_counts_table` property which, if `True`, causes `table.count` to read from the `_counts` table. (#215)
- `table.has_counts_triggers` property revealing if a table has been configured with the new `_counts` database triggers.
- `db.reset_counts()` method and `sqlite-utils reset-counts` command for resetting the values in the `_counts` table. (#219)
- The previously undocumented `db.escape()` method has been renamed to `db.quote()` and is now covered by the documentation: *Quoting strings for use in SQL*. (#217)
- New `table.triggers_dict` and `db.triggers_dict` introspection properties. (#211, #216)
- `sqlite-utils insert` now shows a more useful error message for invalid JSON. (#206)

1.5.18 3.1.1 (2021-01-01)

- Fixed failing test caused by `optimize` sometimes creating larger database files. (#209)
- Documentation now lives on https://sqlite-utils.datasette.io/
- README now includes `brew install sqlite-utils` installation method.

1.5.19 3.1 (2020-12-12)

- New command: `sqlite-utils analyze-tables my.db` outputs useful information about the table columns in the database, such as the number of distinct values and how many rows are null. See *Analyzing tables* for documentation. (#207)
- New `table.analyze_column(column)` Python method used by the `analyze-tables` command - see *Analyzing a column*.
- The `table.update()` method now correctly handles values that should be stored as JSON. Thanks, Andreas Madsack. (#204)

1.5.20 3.0 (2020-11-08)

This release introduces a new `sqlite-utils search` command for searching tables, see *Executing searches*. (#192)

The `table.search()` method has been redesigned, see *Searching with table.search()*. (#197)

The release includes minor backwards-incompatible changes, hence the version bump to 3.0. Those changes, which should not affect most users, are:

- The `-c` shortcut option for outputting CSV is no longer available. The full `--csv` option is required instead.
- The `-f` shortcut for `--fmt` has also been removed - use `--fmt`.
- The `table.search()` method now defaults to sorting by relevance, not sorting by `rowid`. (#198)
- The `table.search()` method now returns a generator over a list of Python dictionaries. It previously returned a list of tuples.

Also in this release:

- The `query`, `tables`, `rows` and `search` CLI commands now accept a new `--tsv` option which outputs the results in TSV. (#193)
- A new `table.virtual_table_using` property reveals if a table is a virtual table, and returns the upper case type of virtual table (e.g. `FTS4` or `FTS5`) if it is. It returns `None` if the table is not a virtual table. (#196)
- The new `table.search_sql()` method returns the SQL for searching a table, see *Building SQL queries with table.search_sql()*.
- `sqlite-utils rows` now accepts multiple optional `-c` parameters specifying the columns to return. (#200)

Changes since the 3.0a0 alpha release:

- The `sqlite-utils search` command now defaults to returning every result, unless you add a `--limit 20` option.
- The `sqlite-utils search -c` and `table.search(columns=[])` options are now fully respected. (#201)

1.5.21 2.23 (2020-10-28)

- `table.m2m(other_table, records)` method now takes any iterable, not just a list or tuple. Thanks, Adam Wolf. (#189)
- `sqlite-utils insert` now displays a progress bar for CSV or TSV imports. (#173)
- New `@db.register_function(deterministic=True)` option for registering deterministic SQLite functions in Python 3.8 or higher. (#191)

1.5.22 2.22 (2020-10-16)

- New `--encoding` option for processing CSV and TSV files that use a non-utf-8 encoding, for both the `insert` and `update` commands. (#182)
- The `--load-extension` option is now available to many more commands. (#137)
- `--load-extension=spatialite` can be used to load SpatiaLite from common installation locations, if it is available. (#136)
- Tests now also run against Python 3.9. (#184)
- Passing `pk=["id"]` now has the same effect as passing `pk="id"`. (#181)

1.5.23 2.21 (2020-09-24)

- `table.extract()` and `sqlite-utils extract` now apply much, much faster - one example operation reduced from twelve minutes to just four seconds! (#172)
- `sqlite-utils extract` no longer shows a progress bar, because it's fast enough not to need one.
- New `column_order=` option for `table.transform()` which can be used to alter the order of columns in a table. (#175)
- `sqlite-utils transform --column-order=` option (with a `-o` shortcut) for changing column order. (#176)
- The `table.transform(drop_foreign_keys=)` parameter and the `sqlite-utils transform --drop-foreign-key` option have changed. They now accept just the name of the column rather than requiring all three of the column, other table and other column. This is technically a backwards-incompatible change but I chose not to bump the major version number because the transform feature is so new. (#177)
- The table `.disable_fts()`, `.rebuild_fts()`, `.delete()`, `.delete_where()` and `.add_missing_columns()` methods all now `return self`, which means they can be chained together with other table operations.

1.5.24 2.20 (2020-09-22)

This release introduces two key new capabilities: **transform** (#114) and **extract** (#42).

Transform

SQLite's ALTER TABLE has several documented limitations. The `table.transform()` Python method and `sqlite-utils transform` CLI command work around these limitations using a pattern where a new table with the desired structure is created, data is copied over to it and the old table is then dropped and replaced by the new one.

You can use these tools to change column types, rename columns, drop columns, add and remove `NOT NULL` and defaults, remove foreign key constraints and more. See the *transforming tables (CLI)* and *transforming tables (Python library)* documentation for full details of how to use them.

Extract

Sometimes a database table - especially one imported from a CSV file - will contain duplicate data. A `Trees` table may include a `Species` column with only a few dozen unique values, when the table itself contains thousands of rows.

The `table.extract()` method and `sqlite-utils extract` commands can extract a column - or multiple columns - out into a separate lookup table, and set up a foreign key relationship from the original table.

The Python library *extract() documentation* describes how extraction works in detail, and *Extracting columns into a separate table* in the CLI documentation includes a detailed example.

Other changes

- The `@db.register_function` decorator can be used to quickly register Python functions as custom SQL functions, see *Registering custom SQL functions*. (#162)
- The `table.rows_where()` method now accepts an optional `select=` argument for specifying which columns should be selected, see *Listing rows*.

1.5.25 2.19 (2020-09-20)

- New `sqlite-utils add-foreign-keys` command for *Adding multiple foreign keys at once*. (#157)
- New `table.enable_fts(..., replace=True)` argument for replacing an existing FTS table with a new configuration. (#160)
- New `table.add_foreign_key(..., ignore=True)` argument for ignoring a foreign key if it already exists. (#112)

1.5.26 2.18 (2020-09-08)

- `table.rebuild_fts()` method for rebuilding a FTS index, see *Rebuilding a full-text search table*. (#155)
- `sqlite-utils rebuild-fts data.db` command for rebuilding FTS indexes across all tables, or just specific tables. (#155)
- `table.optimize()` method no longer deletes junk rows from the `*_fts_docsize` table. This was added in 2.17 but it turns out running `table.rebuild_fts()` is a better solution to this problem.
- Fixed a bug where rows with additional columns that are inserted after the first batch of records could cause an error due to breaking SQLite's maximum number of parameters. Thanks, Simon Wiles. (#145)

1.5.27 2.17 (2020-09-07)

This release handles a bug where replacing rows in FTS tables could result in growing numbers of unnecessary rows in the associated `*_fts_docsize` table. (#149)

- `PRAGMA recursive_triggers=on` by default for all connections. You can turn it off with `Database(recursive_triggers=False)`. (#152)

- `table.optimize()` method now deletes unnecessary rows from the `*_fts_docsize` table. (#153)
- New tracer method for tracking underlying SQL queries, see *Tracing queries*. (#150)
- Neater indentation for schema SQL. (#148)
- Documentation for `sqlite_utils.AlterError` exception thrown by in `add_foreign_keys()`.

1.5.28 2.16.1 (2020-08-28)

- `insert_all(..., alter=True)` now works for columns introduced after the first 100 records. Thanks, Simon Wiles! (#139)
- Continuous Integration is now powered by GitHub Actions. (#143)

1.5.29 2.16 (2020-08-21)

- `--load-extension` option for `sqlite-utils query` for loading SQLite extensions. (#134)
- New `sqlite_utils.utils.find_spatialite()` function for finding SpatiaLite in common locations. (#135)

1.5.30 2.15.1 (2020-08-12)

- Now available as a `sdist` package on PyPI in addition to a wheel. (#133)

1.5.31 2.15 (2020-08-10)

- New `db.enable_wal()` and `db.disable_wal()` methods for enabling and disabling Write-Ahead Logging for a database file - see *WAL mode* in the Python API documentation.
- Also `sqlite-utils enable-wal file.db` and `sqlite-utils disable-wal file.db` commands for doing the same thing on the command-line, see *WAL mode (CLI)*. (#132)

1.5.32 2.14.1 (2020-08-05)

- Documentation improvements.

1.5.33 2.14 (2020-08-01)

- The *insert-files command* can now read from standard input: `cat dog.jpg | sqlite-utils insert-files dogs.db pics - --name=dog.jpg`. (#127)
- You can now specify a full-text search tokenizer using the new `tokenize=` parameter to *enable_fts()*. This means you can enable Porter stemming on a table by running `db["articles"].enable_fts(["headline", "body"], tokenize="porter")`. (#130)
- You can also set a custom tokenizer using the *sqlite-utils enable-fts* CLI command, via the new `--tokenize` option.

1.5.34 2.13 (2020-07-29)

- `memoryview` and `uuid.UUID` objects are now supported. `memoryview` objects will be stored using `BLOB` and `uuid.UUID` objects will be stored using `TEXT`. (#128)

1.5.35 2.12 (2020-07-27)

The theme of this release is better tools for working with binary data. The new `insert-files` command can be used to insert binary files directly into a database table, and other commands have been improved with better support for BLOB columns.

- `sqlite-utils insert-files my.db gifs *.gif` can now insert the contents of files into a specified table. The columns in the table can be customized to include different pieces of metadata derived from the files. See *Inserting binary data from files*. (#122)
- `--raw` option to `sqlite-utils query` - for outputting just a single raw column value - see *Returning raw data, such as binary content*. (#123)
- JSON output now encodes BLOB values as special base64 objects - see *Returning JSON*. (#125)
- The same format of JSON base64 objects can now be used to insert binary data - see *Inserting JSON data*. (#126)
- The `sqlite-utils query` command can now accept named parameters, e.g. `sqlite-utils :memory: "select :num * :num2" -p num 5 -p num2 6` - see *Returning JSON*. (#124)

1.5.36 2.11 (2020-07-08)

- New `--truncate` option to `sqlite-utils insert`, and `truncate=True` argument to `.insert_all()`. Thanks, Thomas Sibley. (#118)
- The `sqlite-utils query` command now runs updates in a transaction. Thanks, Thomas Sibley. (#120)

1.5.37 2.10.1 (2020-06-23)

- Added documentation for the `table.pks` introspection property. (#116)

1.5.38 2.10 (2020-06-12)

- The `sqlite-utils` command now supports UPDATE/INSERT/DELETE in addition to SELECT. (#115)

1.5.39 2.9.1 (2020-05-11)

- Added custom project links to the PyPI listing.

1.5.40 2.9 (2020-05-10)

- New `sqlite-utils drop-table` command, see *Dropping tables*. (#111)
- New `sqlite-utils drop-view` command, see *Dropping views*.
- Python `decimal.Decimal` objects are now stored as `FLOAT`. (#110)

1.5.41 2.8 (2020-05-03)

- New `sqlite-utils create-table` command, see *Creating tables*. (#27)
- New `sqlite-utils create-view` command, see *Creating views*. (#107)

1.5.42 2.7.2 (2020-05-02)

- `db.create_view(...)` now has additional parameters `ignore=True` or `replace=True`, see *Creating views*. (#106)

1.5.43 2.7.1 (2020-05-01)

- New `sqlite-utils views my.db` command for listing views in a database, see *Listing views*. (#105)
- `sqlite-utils tables` (and `views`) has a new `--schema` option which outputs the table/view schema, see *Listing tables*. (#104)
- Nested structures containing invalid JSON values (e.g. Python bytestrings) are now serialized using `repr()` instead of throwing an error. (#102)

1.5.44 2.7 (2020-04-17)

- New `columns=` argument for the `.insert()`, `.insert_all()`, `.upsert()` and `.upsert_all()` methods, for over-riding the auto-detected types for columns and specifying additional columns that should be added when the table is created. See *Custom column order and column types*. (#100)

1.5.45 2.6 (2020-04-15)

- New `table.rows_where(..., order_by="age desc")` argument, see *Listing rows*. (#76)

1.5.46 2.5 (2020-04-12)

- Panda's Timestamp is now stored as a SQLite TEXT column. Thanks, b0b5h4rp13! (#96)
- `table.last_pk` is now only available for inserts or upserts of a single record. (#98)
- New `Database(filepath, recreate=True)` parameter for deleting and recreating the database. (#97)

1.5.47 2.4.4 (2020-03-23)

- Fixed bug where columns with only null values were not correctly created. (#95)

1.5.48 2.4.3 (2020-03-23)

- Column type suggestion code is no longer confused by null values. (#94)

1.5.49 2.4.2 (2020-03-14)

- `table.column_dicts` now works with all column types - previously it would throw errors on types other than `TEXT`, `BLOB`, `INTEGER` or `FLOAT`. (#92)
- Documentation for `NotFoundError` thrown by `table.get(pk)` - see *Retrieving a specific record*.

1.5.50 2.4.1 (2020-03-01)

- `table.enable_fts()` now works with columns that contain spaces. (#90)

1.5.51 2.4 (2020-02-26)

- `table.disable_fts()` can now be used to remove FTS tables and triggers that were created using `table.enable_fts(...)`. (#88)
- The `sqlite-utils disable-fts` command can be used to remove FTS tables and triggers from the command-line. (#88)
- Trying to create table columns with square braces ([or]) in the name now raises an error. (#86)
- Subclasses of `dict`, `list` and `tuple` are now detected as needing a JSON column. (#87)

1.5.52 2.3.1 (2020-02-10)

`table.create_index()` now works for columns that contain spaces. (#85)

1.5.53 2.3 (2020-02-08)

`table.exists()` is now a method, not a property. This was not a documented part of the API before so I'm considering this a non-breaking change. (#83)

1.5.54 2.2.1 (2020-02-06)

Fixed a bug where `.upsert(..., hash_id="pk")` threw an error (#84).

1.5.55 2.2 (2020-02-01)

New feature: `sqlite_utils.suggest_column_types([records])` returns the suggested column types for a list of records. See *Suggesting column types*. (#81).

This replaces the undocumented `table.detect_column_types()` method.

1.5.56 2.1 (2020-01-30)

New feature: `conversions={...}` can be passed to the `.insert()` family of functions to specify SQL conversions that should be applied to values that are being inserted or updated. See *Converting column values using SQL functions* . (#77).

1.5.57 2.0.1 (2020-01-05)

The `.upsert()` and `.upsert_all()` methods now raise a `sqlite_utils.db.PrimaryKeyRequired` exception if you call them without specifying the primary key column using `pk=` (#73).

1.5.58 2.0 (2019-12-29)

This release changes the behaviour of `upsert`. It's a breaking change, hence `2.0`.

The `upsert` command-line utility and the `.upsert()` and `.upsert_all()` Python API methods have had their behaviour altered. They used to completely replace the affected records: now, they update the specified values on existing records but leave other columns unaffected.

See *Upserting data using the Python API* and *Upserting data using the CLI* for full details.

If you want the old behaviour - where records were completely replaced - you can use `$ sqlite-utils insert ... --replace` on the command-line and `.insert(..., replace=True)` and `.insert_all(..., replace=True)` in the Python API. See *Insert-replacing data using the Python API* and *Insert-replacing data using the CLI* for more.

For full background on this change, see issue #66.

1.5.59 1.12.1 (2019-11-06)

- Fixed error thrown when `.insert_all()` and `.upsert_all()` were called with empty lists (#52)

1.5.60 1.12 (2019-11-04)

Python library utilities for deleting records (#62)

- `db["tablename"].delete(4)` to delete by primary key, see *Deleting a specific record*
- `db["tablename"].delete_where("id > ?", [3])` to delete by a where clause, see *Deleting multiple records*

1.5.61 1.11 (2019-09-02)

Option to create triggers to automatically keep FTS tables up-to-date with newly inserted, updated and deleted records. Thanks, Amjith Ramanujam! (#57)

- `sqlite-utils enable-fts ... --create-triggers` - see *Configuring full-text search using the CLI*
- `db["tablename"].enable_fts(..., create_triggers=True)` - see *Configuring full-text search using the Python library*
- Support for introspecting triggers for a database or table - see *Introspection* (#59)

1.5.62 1.10 (2019-08-23)

Ability to introspect and run queries against views (#54)

- `db.view_names()` method and and `db.views` property
- Separate `View` and `Table` classes, both subclassing new `Queryable` class

- `view.drop()` method

See *Listing views*.

1.5.63 1.9 (2019-08-04)

- `table.m2m(...)` method for creating many-to-many relationships: *Working with many-to-many relationships* (#23)

1.5.64 1.8 (2019-07-28)

- `table.update(pk, values)` method: *Updating a specific record* (#35)

1.5.65 1.7.1 (2019-07-28)

- Fixed bug where inserting records with 11 columns in a batch of 100 triggered a "too many SQL variables" error (#50)
- Documentation and tests for `table.drop()` method: *Dropping a table or view*

1.5.66 1.7 (2019-07-24)

Support for lookup tables.

- New `table.lookup({...})` utility method for building and querying lookup tables - see *Working with lookup tables* (#44)
- New `extracts=` table configuration option, see *Populating lookup tables automatically during insert/upsert* (#46)
- Use pysqlite3 if it is available, otherwise use `sqlite3` from the standard library
- Table options can now be passed to the new `db.table(name, **options)` factory function in addition to being passed to `insert_all(records, **options)` and friends - see *Table configuration options*
- In-memory databases can now be created using `db = Database(memory=True)`

1.5.67 1.6 (2019-07-18)

- `sqlite-utils insert` can now accept TSV data via the new `--tsv` option (#41)

1.5.68 1.5 (2019-07-14)

- Support for compound primary keys (#36)
  - Configure these using the CLI tool by passing `--pk` multiple times
  - In Python, pass a tuple of columns to the `pk=(..., ...)` argument: *Compound primary keys*
- New `table.get()` method for retrieving a record by its primary key: *Retrieving a specific record* (#39)

1.5.69 1.4.1 (2019-07-14)

- Assorted minor documentation fixes: changes since 1.4

1.5.70 1.4 (2019-06-30)

- Added `sqlite-utils index-foreign-keys` command (*docs*) and `db.index_foreign_keys()` method (*docs*) (#33)

1.5.71 1.3 (2019-06-28)

- New mechanism for adding multiple foreign key constraints at once: *db.add_foreign_keys() documentation* (#31)

1.5.72 1.2.2 (2019-06-25)

- Fixed bug where `datetime.time` was not being handled correctly

1.5.73 1.2.1 (2019-06-20)

- Check the column exists before attempting to add a foreign key (#29)

1.5.74 1.2 (2019-06-12)

- Improved foreign key definitions: you no longer need to specify the `column`, `other_table` AND `other_column` to define a foreign key - if you omit the `other_table` or `other_column` the script will attempt to guess the correct values by introspecting the database. See *Adding foreign key constraints* for details. (#25)
- Ability to set `NOT NULL` constraints and `DEFAULT` values when creating tables (#24). Documentation: *Setting defaults and not null constraints (Python API)*, *Setting defaults and not null constraints (CLI)*
- Support for `not_null_default=X` / `--not-null-default` for setting a `NOT NULL DEFAULT 'x'` when adding a new column. Documentation: *Adding columns (Python API)*, *Adding columns (CLI)*

1.5.75 1.1 (2019-05-28)

- Support for `ignore=True` / `--ignore` for ignoring inserted records if the primary key already exists (#21) - documentation: *Inserting data (Python API)*, *Inserting data (CLI)*
- Ability to add a column that is a foreign key reference using `fk=...` / `--fk` (#16) - documentation: *Adding columns (Python API)*, *Adding columns (CLI)*

1.5.76 1.0.1 (2019-05-27)

- `sqlite-utils rows data.db table --json-cols` - fixed bug where `--json-cols` was not obeyed

1.5.77 1.0 (2019-05-24)

- **Option to automatically add new columns if you attempt to insert or upsert data with extra fields:**
  `sqlite-utils insert ... --alter` - see *Adding columns automatically with the sqlite-utils CLI*

  `db["tablename"].insert(record, alter=True)` - see *Adding columns automatically using the Python API*
- New `--json-cols` option for outputting nested JSON, see *Nested JSON values*

1.5.78 0.14 (2019-02-24)

- Ability to create unique indexes: `db["mytable"].create_index(["name"], unique=True)`
- `db["mytable"].create_index(["name"], if_not_exists=True)`
- `$ sqlite-utils create-index mydb.db mytable col1 [col2...]`, see *Creating indexes*
- `table.add_column(name, type)` method, see *Adding columns*
- `$ sqlite-utils add-column mydb.db mytable nameofcolumn`, see *Adding columns* (CLI)
- `db["books"].add_foreign_key("author_id", "authors", "id")`, see *Adding foreign key constraints*
- `$ sqlite-utils add-foreign-key books.db books author_id authors id`, see *Adding foreign key constraints* (CLI)
- Improved (but backwards-incompatible) `foreign_keys=` argument to various methods, see *Specifying foreign keys*

1.5.79 0.13 (2019-02-23)

- New `--table` and `--fmt` options can be used to output query results in a variety of visual table formats, see *Table-formatted output*
- New `hash_id=` argument can now be used for *Setting an ID based on the hash of the row contents*
- Can now derive correct column types for numpy int, uint and float values
- `table.last_id` has been renamed to `table.last_rowid`
- `table.last_pk` now contains the last inserted primary key, if `pk=` was specified
- Prettier indentation in the `CREATE TABLE` generated schemas

1.5.80 0.12 (2019-02-22)

- Added `db[table].rows` iterator - see *Listing rows*
- Replaced `sqlite-utils json` and `sqlite-utils csv` with a new default subcommand called `sqlite-utils query` which defaults to JSON and takes formatting options `--nl`, `--csv` and `--no-headers` - see *Returning JSON* and *Returning CSV or TSV*
- New `sqlite-utils rows data.db name-of-table` command, see *Returning all rows in a table*
- `sqlite-utils table` command now takes options `--counts` and `--columns` plus the standard output format options, see *Listing tables*

1.5.81 0.11 (2019-02-07)

New commands for enabling FTS against a table and columns:

```
sqlite-utils enable-fts db.db mytable col1 col2
```

See *Configuring full-text search*.

1.5.82 0.10 (2019-02-06)

Handle `datetime.date` and `datetime.time` values.

New option for efficiently inserting rows from a CSV:

```
sqlite-utils insert db.db foo - --csv
```

1.5.83 0.9 (2019-01-27)

Improved support for newline-delimited JSON.

`sqlite-utils insert` has two new command-line options:

- `--nl` means "expect newline-delimited JSON". This is an extremely efficient way of loading in large amounts of data, especially if you pipe it into standard input.
- `--batch-size=1000` lets you increase the batch size (default is 100). A commit will be issued every X records. This also control how many initial records are considered when detecting the desired SQL table schema for the data.

In the Python API, the `table.insert_all(...)` method can now accept a generator as well as a list of objects. This will be efficiently used to populate the table no matter how many records are produced by the generator.

The `Database()` constructor can now accept a `pathlib.Path` object in addition to a string or an existing SQLite connection object.

1.5.84 0.8 (2019-01-25)

Two new commands: `sqlite-utils csv` and `sqlite-utils json`

These commands execute a SQL query and return the results as CSV or JSON. See *Returning CSV or TSV* and *Returning JSON* for more details.

```
$ sqlite-utils json --help
Usage: sqlite-utils json [OPTIONS] PATH SQL

  Execute SQL query and return the results as JSON

Options:
  --nl       Output newline-delimited JSON
  --arrays   Output rows as arrays instead of objects
  --help     Show this message and exit.

$ sqlite-utils csv --help
Usage: sqlite-utils csv [OPTIONS] PATH SQL
```

(continues on next page)

(continued from previous page)

```
  Execute SQL query and return the results as CSV

Options:
  --no-headers  Exclude headers from CSV output
  --help        Show this message and exit.
```

1.5.85 0.7 (2019-01-24)

This release implements the `sqlite-utils` command-line tool with a number of useful subcommands.

- `sqlite-utils tables demo.db` lists the tables in the database
- `sqlite-utils tables demo.db --fts4` shows just the FTS4 tables
- `sqlite-utils tables demo.db --fts5` shows just the FTS5 tables
- `sqlite-utils vacuum demo.db` runs VACUUM against the database
- `sqlite-utils optimize demo.db` runs OPTIMIZE against all FTS tables, then VACUUM
- `sqlite-utils optimize demo.db --no-vacuum` runs OPTIMIZE but skips VACUUM

The two most useful subcommands are `upsert` and `insert`, which allow you to ingest JSON files with one or more records in them, creating the corresponding table with the correct columns if it does not already exist. See *Inserting JSON data* for more details.

- `sqlite-utils insert demo.db dogs dogs.json --pk=id` inserts new records from `dogs.json` into the `dogs` table
- `sqlite-utils upsert demo.db dogs dogs.json --pk=id` upserts records, replacing any records with duplicate primary keys

One backwards incompatible change: the `db["table"].table_names` property is now a method:

- `db["table"].table_names()` returns a list of table names
- `db["table"].table_names(fts4=True)` returns a list of just the FTS4 tables
- `db["table"].table_names(fts5=True)` returns a list of just the FTS5 tables

A few other changes:

- Plenty of updated documentation, including full coverage of the new command-line tool
- Allow column names to be reserved words (use correct SQL escaping)
- Added automatic column support for bytes and datetime.datetime

1.5.86 0.6 (2018-08-12)

- `.enable_fts()` now takes optional argument `fts_version`, defaults to `FTS5`. Use `FTS4` if the version of SQLite bundled with your Python does not support FTS5
- New optional `column_order=` argument to `.insert()` and friends for providing a partial or full desired order of the columns when a database table is created
- *New documentation* for `.insert_all()` and `.upsert()` and `.upsert_all()`

1.5.87 0.5 (2018-08-05)

- `db.tables` and `db.table_names` introspection properties
- `db.indexes` property for introspecting indexes
- `table.create_index(columns, index_name)` method
- `db.create_view(name, sql)` method
- Table methods can now be chained, plus added `table.last_id` for accessing the last inserted row ID

1.5.88 0.4 (2018-07-31)

- `enable_fts()`, `populate_fts()` and `search()` table methods

Take a look at this script for an example of this library in action.