ConvertCSV Using Node.js CSV-Parser

Convert CSV files into data used by SQLite3 (Version 0.0.73)

LOLH

CSV-SQLite3 Using Node Version 0.0.73 (Fri 10-18-2019)

Table of Contents

Problem Statement **1**
The Workflow 1
Source Code for catuniq.pl 3

Introduction **6**

1 SQLite Tables **7**
1.1 SQLite USB Table Columns 7
1.2 SQLite Checks Table Columns 9

2 CSV-SQLite3 Usage **11**

3 Create the Project **12**
3.1 Install the CSV-SQLite3 Package and Dependencies 12
3.2 Import the Node.js Dependencies 16
3.3 Establish Database Table Name and Columns 17

4 Working with the Command Line **19**
4.1 Command Line Usage 19
4.2 Command Line Argument Processing 19

5 Attach To or Delete a Database **21**

6 Find and Store Checks **26**

7 Export SQLite DB Data to CSV File **29**
7.1 '`--export`' Option 29
7.2 Ledger `convert` Command 29
7.3 Export Code 29
7.4 The Zero Ledger File 32
7.5 The Accounts Payable File 35

8 Process CSV Files **36**
8.1 Set Up CSV-Stringify 36
8.2 Set Up Stream-Transform and Transform Function 38
8.2.1 Set Up the Transform Function 38
8.2.2 Set Up the Stream Transform 42
8.3 Set Up CSV-Parse 43
8.4 Set Up StreamReader 43

Appendix A Node-SQLite3 Module 46
A.1 Node-SQLite3 Module Usage 46
A.2 Features 46
A.3 Node-SQLite3 API 46
A.4 Node-SQLite3 Control Flow—Two Execution Modes 47
A.4.1 Serialize Execution Mode 47
A.4.2 Parallelize Execution Mode 48

Appendix B Converting CSV Files 49
B.1 Ledger Convert Command 49

9 Update package.json Version 52

10 Makefile 53

Index 56

Function Index 58

Listings 59

Problem Statement

Instead of using Intuit Quicken or Microsoft Money, or even GnuCash, I would like to use Ledger3 to manage my finances. Two big problems immediately arise:

1. What do I do about the years of prior data that I have accumulated?
2. How do I efficiently process new data as it comes in from my bank?

The prior data comes in many different sizes and styles. For the immediate future, I will focus solely on the years 2016-2019. I can add additional years after getting things to work for 2016-2019.

New data (transactions) come in from my bank through CSV and QFX downloads. I will download and save both of the CSV and QFX data, but work exclusively with the CSV files at this point.

There is a '`Download Transactions`' button that loads a modal window with settings for

- Account
- From Date
- To Date
- Download To (i.e., format)

So I can easily pick the account, start date, end date, and format, download new transactions into a separate file. The problem that arises is that because this is a manual operation, it becomes difficult to keep track of what dates have been downloaded and processed already, or even wanting to reprocess some dates or all dates, etc. without accumulating duplicate data. An acceptable solution is to concatenate newly downloaded data into files according to account and year, then remove duplicate items. This would be accomplished with a Perl script concatenating the files, sorting the lines, and unique'ifying them and then saving and backing up new and original files.

The Workflow

A potential workflow would therefore be:

1. Download raw data from a bank by account, start date, and end date, which will be stored in the '`~/Downloads`' directory as `export.csv`. A script can be executed to review the data to determine what the year is. However, there is no indication in either the filename or the data from which account the data was downloaded. Therefore, any script that is run must be told what account this data is associated with, as for example here:

   ```
   catuniq [--dest <path>] <acct>
   ```

   Invoking this command will look for a file name `export.csv` in the '`~/Downloads`' directory, determine what year its transactions are, and catenate the file onto the proper destination file according to year and account, and then sort and unique'ify the new file. It will create the necessary destination directory and file to allow the command to run.

   The file onto which the export is catenated can either be indicated on the command line by means of the `--dest` option, or located through the environment variable '`$WORKUSB`'

if the `--dest` option is omitted. Backups are stored in $WORKBAK with the Unix time appended to the end.

Perl script catuniq.pl

2. Process the raw data to remove extraneous and unnecessary data while reformating the data into the proper form and columns for efficiently working with the financial data, e.g., producing accounting and tax reports, budgets, etc.. This processed data will be stored in an SQLite3 database, avoiding duplicating data at all times.

3. Export data from the SQLite3 database, and convert it into a form usable by the Ledger3 program, again always avoiding duplicating transactions.

4. Do as much as possible with the Ledger3 data automatically to create proper double entry transactions to avoid time-consuming manual work.

This workflow requires a way to download any amount of data and process it with the click of a button without worrying about duplicating transactions. Ledger3 is supposed to be able to detect duplicates and avoid reloading them with its `convert` command. Therefore it would be convenient to be able to download any amount of new data from a bank and have it combined with old data without duplicating transactions as well.

Then, the workflow would process the downloaded data into a form usable in both an SQLite3 database and Ledger3, again without having to worry about duplicating data. Ledger3 uses a hashing function to produce a unique hash for each entry, similar to how `git` works. This may be a possible solution for the raw bank data as well.

Source Code for catuniq.pl

```
 1
 2  package CATUNIQ;
 3  our $VERSION = "0.1.2";
 4  eval $VERSION;
 5  # 2019-07-31T12:30
 6
 7  use strict;
 8  use warnings; no warnings 'experimental';
 9  use v5.16;
10
11  use File::Spec;                         # Portably perform operations on file name
12  use File::Path;                         # Create or remove directory trees
13  use File::Slurp;                # Simple and Efficient Reading/Writing/Modifying
14  use File::Copy;                         # Copy files or filehandles
15  use List::Util 'uniqstr';        # A selection of general-utility list subroutine
16  use OptArgs2;                       # Integrated argument and option processing
17  use Data::Printer;                # colored pretty-print of Perl data structures
18
19
20  my @valid_accts = (qw'
21      6815
22      6831
23      6151'
24  );
25
26  my @valid_years = (qw'
27      2016
28      2017
29      2018
30      2019'
31  );
32
33
34  # ESTABLISH THE CL OPTIONS AND ARGUMENTS
35  opt help => (
36      isa => 'Flag',
37      comment => 'Help',
38      alias => 'h',
39      ishelp => 1,
40  );
41
42  arg acct => (
43      isa     => 'Str',
44      comment => 'The name of the account to which the file is related; e.g. "usb_6
45      required=> 1,
46  );
47
48  opt dest => (
49      isa     => 'Str',
50      alias   => 'd',
51      comment => 'path to the destination file upon which the new data will be cate
```

```
ln -f $PWD/scripts/catuniq.pl $WORKBIN/catuniq
```

Introduction

US Bank has the facility to download bank records in CSV form. This program is designed to convert those downloaded CSV files into a form usable by SQLite, and then to use SQLite to process the data.

Header Lines

The header lines are:

```
"Date","Transaction","Name","Memo","Amount"
```

Data Columns

Date

A sample date is:

```
1/4/2016
```

This should be transformed into:

```
2016-01-14
```

Transaction

A '`Transaction`' is one of

- '`DEBIT`'
- '`CREDIT`'

This should be transformed into:

```
debit | credit
```

Name

A sample '`name`' entry is:

```
DEBIT PURCHASE -VISA USPS PO BOXES 66800-3447779 DC
```

This should be transformed into:

```
visa purchase usps po boxes
```

Memo

A sample '`memo`' entry is:

```
Download from usbank.com. USPS PO BOXES 66800-3447779 DC
```

The '`Download from usbank.com`' should be removed in all cases. The extraneous numbers should be removed whenever possible.

Amount

- -66.0000

This should be transformed into:

```
$-66.00
```

1 SQLite Tables

The minimum SQLite tables that should be created are:

usb	includes business (6815), trust (6831), personal (6151) data
cases	190301, etc.
people	John Doe, Mary Jane, etc.
businesses	Law Office of . . . , etc.
checks	holds check information parsed from `worklog.<year>.otl` files

More can be created as needed.

1.1 SQLite USB Table Columns

The columns that should be created for the usb bank tables are:

- '`rowid`' (implicit creation)
- '`acct`' in the form of '`usb_6815|usb_6831|usb_6151`'
- '`date`' in the form of '`yyyy-mm-dd`'
- '`trans`' containing either '`CREDIT | DEBIT`'
- '`checkno`' containing a check number, if present ('`check`' is a reserved word and throws an error)
- '`txfr`' containing a direction arrow ('`<`' or '`>`') and a bank account ('`usb_6151`')
- '`payee`'
- '`category`'
- '`memo`'
- '`desc1`'
- '`desc2`'
- '`caseno`' containing a related case number ('`case`' is apparently a reserved word and throws an error)
- '`amount`' in the form '`\pm##,###.##`'

rowid	**acct**	**date**	**trans**	**checkno**	**txfr**	**payee**	**category**	**me**
primary key	'`usb_6815`'	yyyy-mm-dd	credit	####	< usb 6151	text	text	tex
implicit creation	'`usb_6831`' '`usb_6151`'	not null	debit	null	> usb 6831 null	not null	null	nul

Table 1.1: USB Table Column Names

```
usb (
       rowid     INTEGER PRIMARY KEY NOT NULL,
       acct      TEXT NOT NULL,
       date      TEXT NOT NULL,
       trans     TEXT NOT NULL,
       checkno   TEXT,
       txfr      TEXT,
       payee     TEXT NOT NULL,
       category  TEXT,
       note      TEXT,
       desc1     TEXT,
       desc2     TEXT,
       caseno    TEXT,
       amount    REAL NOT NULL,
       OrigPayee TEXT NOT NULL,
       OrigMemo  TEXT NOT NULL )
```

Listing 1.1: Define the USB Table Schema

The database column names are identical to the SQL column names except for '`rowid`', which is missing since it is auto-generated by SQLite3.

```
const DB_TABLES = {
    usb: 'usb',
    checks: 'checks',
};

const DB_ACCTS = {
    '6815': 'Business',
    '6831': 'Trust',
    '6151': 'Personal',
};

const DB_YEARS = [
    '2016',
    '2017',
    '2018',
]

const DB_COLS = [
    'acct',
    'date',
    'trans',
    'checkno',
    'txfr',
    'payee',
    'category',
    'note',
```

```
        'desc1',
        'desc2',
        'caseno',
        'amount',
        'OrigPayee',
        'OrigMemo',
    ];

    const EXPORT_DB_COLS = [
        'rowid',
        'acct',
        'date',
        'trans',
        'checkno',
        'txfr',
        'payee',
        'category',
        'note',
        'caseno',
        'amount',
    ];
```

1.2 SQLite Checks Table Columns

The column names for the '`checks`' table columns are:

- '`rowid`'
- '`acct`' e.g., 6815, 6831, 6151
- '`checkno`', e.g., 1001, 1002
- '`date`'
- '`payee`'
- '`subject`'
- '`purpose`'
- '`caseno`', e.g., 190301, 190205
- '`amount`'

rowid	**acct**	**checkno**	**date**	**payee**	**subject**	**purpose**	**caseno**	**amount**
INTEGER	TEXT	INTEGER	TEXT	TEXT	TEXT	TEXT	TEXT	REAL
PRIMARY	NOT	NOT	NOT	NOT	NOT			NOT
KEY	NULL	NULL	NULL	NULL	NULL			NULL

Table 1.2: Checks Table Columns and Data Types

```
checks (
      acct                TEXT NOT NULL,
      checkno               TEXT NOT NULL,
      date                TEXT NOT NULL,
      payee                TEXT NOT NULL,
      subject               TEXT NOT NULL,
      purpose               TEXT,
      caseno               TEXT NOT NULL,
      amount               REAL NOT NULL
)
```

Listing 1.2: Define Checks Table Schema

```
const CHECKS_COLS = [
    'acct',
    'checkno',
    'date',
    'payee',
    'subject',
    'purpose',
    'caseno',
    'amount'
];
```

2 CSV-SQLite3 Usage

- ~csv-sqlite3 –csv <usb-acct> <year> [–attach <db>] [–checks]
  - <usb-acct> is one of: '6815|6831|6151'
  - <year> is one of '2016|2017|2018'
  - --attach <db> is optional; the default is workfin.sqlite, otherwise <db>.sqlite
  - --checks parses worklog.<year>.otl files for check data and places it into <db> in a 'checks' documents
  - Transform downloaded data and save in a new CSV file and a new sqlite3 database. The source CSV files are found at:

    $WORKFIN/src/usb/usb_{6815|6831|6151}/{2016|2017|2018}/usb_6815--2016.csv
  - The transformed CSV files are found at:

    $WORKFIN/csv/usb_{6815|6831|6151}__{2016|2017|2018}.csv
  - The new SQLite3 database is found at:

    $WORKFIN/db/workfin.sqlite
  - The exported database file in CSV format will be found at:

    $WORKFIN/csv/workfin.csv
- csv-sqlite3 --delete [<db>] Delete the SQLite3 database file <db>.sqlite, or workfin.sqlite if left blank; also delete the CSV files in csv/;
- csv-sqlite3 --log-level <level>

  Set a log level (default is 'warn')

3 Create the Project

External Dependencies

This project's external dependencies are the following Node.js modules:

`command-line-args`
: `https://github.com/75lb/command-line-args#readme`

`command-line-usage`
: `https://github.com/75lb/command-line-usage`

`csv`
: `https://csv.js.org/`

`sqlite3`
: `https://github.com/mapbox/node-sqlite3/wiki`

`accounting`
: `http://openexchangerates.github.io/accounting.js/`

Node.js Dependencies

This project's Node.js internal dependencies are the following:

`File System (fs)`
: File System for working with files

`Path (path)`
: Utilities for working with file and directory paths

`Utilities (util)`
: Utilities for inspecting objects

`Assert (assert)`
: Assertion functions for verifying invariants

3.1 Install the CSV-SQLite3 Package and Dependencies

Create the CSV-SQLite3 project and install the Node.js package dependencies:

- `command-line-ars`
- `command-line-usage`
- `csv`
- `sqlite3`
- `accounting`

```
yarn --yes --private init
sed -i '' -e 's/\"version\": \"1.0.0\",/\"version\": \"0.0.73\",/' package.json
yarn add command-line-args command-line-usage csv sqlite3 accounting
yarn add ssh://git@github.com:wlharvey4/wlparser.git#prod
```

Listing 3.1: Create the CSV-SQLite3 Project

```
yarn init v1.19.0
```

```
success Saved package.json
Done in 0.04s.
yarn add v1.19.0
info No lockfile found.
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...
success Saved lockfile.
success Saved 130 new dependencies.
info Direct dependencies
 accounting@0.4.1
 command-line-args@5.1.1
 command-line-usage@6.0.2
 csv@5.1.3
 sqlite3@4.1.0
info All dependencies
 abbrev@1.1.1
 accounting@0.4.1
 ajv@6.10.2
 ansi-regex@2.1.1
 ansi-styles@3.2.1
 aproba@1.2.0
 are-we-there-yet@1.1.5
 asn1@0.2.4
 asynckit@0.4.0
 aws-sign2@0.7.0
 aws4@1.8.0
 balanced-match@1.0.0
 bcrypt-pbkdf@1.0.2
 brace-expansion@1.1.11
 caseless@0.12.0
 chalk@2.4.2
 chownr@1.1.3
 clone@1.0.4
 code-point-at@1.1.0
 color-convert@1.9.3
 color-name@1.1.3
 combined-stream@1.0.8
 command-line-args@5.1.1
 command-line-usage@6.0.2
 concat-map@0.0.1
 console-control-strings@1.1.0
 core-util-is@1.0.2
 csv-generate@3.2.3
 csv-parse@4.6.3
 csv-stringify@5.3.3
```

```
csv@5.1.3
dashdash@1.14.1
debug@3.2.6
deep-extend@0.6.0
defaults@1.0.3
delayed-stream@1.0.0
delegates@1.0.0
detect-libc@1.0.3
ecc-jsbn@0.1.2
escape-string-regexp@1.0.5
extend@3.0.2
extsprintf@1.3.0
fast-deep-equal@2.0.1
fast-json-stable-stringify@2.0.0
find-replace@3.0.0
forever-agent@0.6.1
form-data@2.3.3
fs-minipass@1.2.7
fs.realpath@1.0.0
gauge@2.7.4
getpass@0.1.7
glob@7.1.4
har-schema@2.0.0
har-validator@5.1.3
has-flag@3.0.0
has-unicode@2.0.1
http-signature@1.2.0
iconv-lite@0.4.24
ignore-walk@3.0.3
inflight@1.0.6
inherits@2.0.4
ini@1.3.5
is-fullwidth-code-point@1.0.0
is-typedarray@1.0.0
isarray@1.0.0
isstream@0.1.2
json-schema-traverse@0.4.1
json-schema@0.2.3
json-stringify-safe@5.0.1
jsprim@1.4.1
lodash.camelcase@4.3.0
mime-db@1.40.0
mime-types@2.1.24
minimist@0.0.8
minipass@2.9.0
minizlib@1.3.3
mixme@0.3.2
```

```
mkdirp@0.5.1
ms@2.1.2
nan@2.14.0
needle@2.4.0
node-pre-gyp@0.11.0
nopt@4.0.1
npm-bundled@1.0.6
npm-packlist@1.4.6
npmlog@4.1.2
number-is-nan@1.0.1
oauth-sign@0.9.0
object-assign@4.1.1
os-homedir@1.0.2
os-tmpdir@1.0.2
osenv@0.1.5
pad@3.2.0
path-is-absolute@1.0.1
performance-now@2.1.0
process-nextick-args@2.0.1
psl@1.4.0
punycode@1.4.1
qs@6.5.2
rc@1.2.8
readable-stream@2.3.6
reduce-flatten@2.0.0
request@2.88.0
rimraf@2.7.1
safer-buffer@2.1.2
sax@1.2.4
semver@5.7.1
set-blocking@2.0.0
signal-exit@3.0.2
sqlite3@4.1.0
sshpk@1.16.1
stream-transform@2.0.1
string_decoder@1.1.1
string-width@1.0.2
strip-ansi@3.0.1
strip-json-comments@2.0.1
supports-color@5.5.0
table-layout@1.0.0
tar@4.4.13
tough-cookie@2.4.3
tunnel-agent@0.6.0
tweetnacl@0.14.5
uri-js@4.2.2
util-deprecate@1.0.2
```

```
 uuid@3.3.3
 verror@1.10.0
 wcwidth@1.0.1
 wide-align@1.1.3
 wordwrapjs@4.0.0
 yallist@3.1.1
Done in 4.71s.
yarn add v1.19.0
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...
success Saved lockfile.
success Saved 1 new dependency.
info Direct dependencies
 wlparser@0.3.1
info All dependencies
 wlparser@0.3.1
Done in 2.72s.

cat package.json
```

Listing 3.2: Package json

```
{
  "name": "csv-sqlite3",
  "version": "0.0.73",
  "main": "index.js",
  "repository": "ssh://git@github.com/wlharvey4/csv-sqlite3.git",
  "author": "Wesley Harvey <pinecone062@gmail.com>",
  "license": "MIT",
  "private": true,
  "dependencies": {
    "accounting": "^0.4.1",
    "command-line-args": "^5.1.1",
    "command-line-usage": "^6.0.2",
    "csv": "^5.1.3",
    "sqlite3": "^4.1.0",
    "wlparser": "ssh://git@github.com:wlharvey4/wlparser.git#prod"
  }
}
```

3.2 Import the Node.js Dependencies

Create the base file `index.js` and import the project's dependencies.

```
1  /* index.js */
2
3  const fs     = require('fs');
4  const path   = require('path');
```

```
 5  const util   = require('util');
 6  const assert = require('assert').strict;
 7
 8  const cl_args  = require('command-line-args');
 9  const cl_usage = require('command-line-usage');
10  const csv      = require('csv');
11  const sqlite3  = require('sqlite3').verbose();      // remove 'verbose' in product
12  const accounting = require('accounting');
```

3.3 Establish Database Table Name and Columns

```
13  const DB_TABLES = {
14      usb: 'usb',
15      checks: 'checks',
16  };
17
18  const DB_ACCTS = {
19      '6815': 'Business',
20      '6831': 'Trust',
21      '6151': 'Personal',
22  };
23
24  const DB_YEARS = [
25      '2016',
26      '2017',
27      '2018',
28  ]
29
30  const DB_COLS = [
31      'acct',
32      'date',
33      'trans',
34      'checkno',
35      'txfr',
36      'payee',
37      'category',
38      'note',
39      'desc1',
40      'desc2',
41      'caseno',
42      'amount',
43      'OrigPayee',
44      'OrigMemo',
45  ];
46
47  const EXPORT_DB_COLS = [
48      'rowid',
```

```
49      'acct',
50      'date',
51      'trans',
52      'checkno',
53      'txfr',
54      'payee',
55      'category',
56      'note',
57      'caseno',
58      'amount',
59  ];
60
61  const CHECKS_COLS = [
62      'acct',
63      'checkno',
64      'date',
65      'payee',
66      'subject',
67      'purpose',
68      'caseno',
69      'amount'
70  ];
```

4 Working with the Command Line

Here is implementation of command-line argument parsing and the generation of a usage message triggered by requesting the option '`--help`'.

4.1 Command Line Usage

This section generates a usage message activated by the '`--help`' option. It uses the [`options_defs`], page 20, object created in the code in the next section.

```
const sections = [
    {
        header: 'CSV-SQLite3',
        content: 'Processes raw usb csv files into a form usable by SQLite3'
    },
    {
        header: 'Options',
        optionList: option_defs,
    },
    {
        content: `Project directory: {underline ${process.env.WORKNODE}/CSV-SQLite3}`
    }
];
const usage = cl_usage(sections);
console.log(usage);
```

4.2 Command Line Argument Processing

Options include giving the name of a database to attach to using '`--attach <db>`'. In the absence of this option, a default database will be used. A database can be deleted here as well using the option '`--delete <db>`', with a backup being saved in the '`WORKBAK`' directory wtih the unix time suffixed to the end. If the option `--checks` is included, parse the `worklog.<year>.otl` worklog file for check information and save the same in a '`checks`' table in '`<db>`'.

Usage

Obtain usage information using the `--help` option:

```
csv-sqlite3 --help | -h
```

Choose the CSV File

For non-delete commands, identify the CSV file to transform via the '`--csv`' option:

```
csv-sqlite3 --csv | -c 6815|6831 2004...2019
```

Attach and Delete a Database

The database is attachable (it will be created automatically if it does not exist), and deleteable:

```
csv-sqlite3 --attach <db> | -a <db-name>
csv-sqlite3 --delete <db> | -d <db-name>
```

Export the SQLite3 data to CSV

Export SQLite3 data into a CSV file of the same name:

```
csv-sqlite3 --export <db> | -e <db-name>
```

Set a Log Level

```
csv-sqlite3 --log-level <value:0..10> | -l <value:0..10>
```

```
 71  const option_defs = [
 72      { name: 'help',   alias: 'h', type: Boolean, description: 'Prints this usage
 73      { name: 'attach', alias: 'a', type: String,  description: 'Attach to an exist
 74      { name: 'delete', alias: 'd', type: String,  description: 'Delete an existing
 75      { name: 'csv',    alias: 'c', type: String,  description: 'Process a CSV file
 76      { name: 'export', alias: 'e', type: String,  description: 'Export sqlite3 dat
 77      { name: 'checks',             type: Boolean, description: 'Find checks in wor
 78      { name: 'log-level', alias: 'l', type: Number, description: 'Set a log level
 79  ];
 80  const options = cl_args(option_defs);
 81  console.log(options);
 82
 83  if (options.help) {
 84      const sections = [
 85          {
 86              header: 'CSV-SQLite3',
 87              content: 'Processes raw usb csv files into a form usable by SQLite3'
 88          },
 89          {
 90              header: 'Options',
 91              optionList: option_defs,
 92          },
 93          {
 94              content: `Project directory: {underline ${process.env.WORKNODE}/CSV-S
 95          }
 96      ];
 97      const usage = cl_usage(sections);
 98      console.log(usage);
 99      process.exit(0);
100  }
101
102  let LOG_LEVEL = process.env.LOG_LEVEL || 1;
103  if (options['log-level'] >= 0) {
104      if (typeof options['log-level'] === 'number' && options['log-level'] <= 10)
105          LOG_LEVEL = options['log-level'];
106      else {
107          console.error(`Incorrect log-level: ${options['log-level']}; must be betw
108      }
109  }
110  console.log(`Log-level set at: ${LOG_LEVEL}`);
```

5 Attach To or Delete a Database

SQLite3 can have any number of databases. Only one is initially attached, but more can be attached subsequent to the first attachment. If the database does not exist, it will be created. If the user requests that a database file be deleted, it will be backed up first, then deleted.

The user can attach to a database file (either a specified file or the default file, defined as `$WORKFIN/db/workfin.sqlite`), or delete a specified database file and the associated CSV file exported from the database data. All deleted files are backed up to a backup directory that needs to be defined as a shell environment variable: '`WORKBAK`'. The backed-up files are appended with the current date in UNIX time (seconds).

The attached database will be referenced as `db`.

Verbose Mode

During development, call the `verbose()` method on the `sqlite3` object to enable better stack traces. In production, remove this call for improved performance.

```
111
112 if ( !process.env.WORKDB ) { // $WORKFIN/db
113     console.error('You must define a shell variable named WORKDB as a base direct
114     process.exit(1);
115 }
116 if ( !process.env.WORKCSV ) { // $WORKFIN/csv
117     console.error('You must define a shell variable named WORKCSV as a base direc
118     process.exit(1);
119 }
120 if ( !process.env.WORKLEDGER ) { // $WORKFIN/ledger
121     console.error('You must define a shell variable named WORKLEDGER as a base di
122     process.exit(1);
123 }
124 if ( !process.env.WORKBAK) { // $WORK/workbak
125     console.error('You must define a shell variable named WORKBAK as a backup dir
126     process.exit(1);
127 }
128
129 const WORKDB     = process.env.WORKDB;     // base directory for .sqlite db files
130 if (!fs.existsSync(WORKDB)) { fs.mkdirSync(WORKDB); }
131 const WORKCSV    = process.env.WORKCSV;    // base directory for .csv files
132 if (!fs.existsSync(WORKCSV)) { fs.mkdirSync(WORKCSV); }
133 const WORKLEDGER = process.env.WORKLEDGER; // base directory for .ledger files
134 const WORKBAK    = process.env.WORKBAK;    // base directory for storing deleted
135
136 const DB_DEFAULT = 'workfin';               // default sqlite db name
137 const db_file = options.attach ? options.attach :    // use provided option for a
138                 options.delete ? options.delete :    // use provided option for d
139                 DB_DEFAULT;                                  // if no provided op
```

```
140
141 const db_path = path.format({
142     dir: WORKDB,
143     name: db_file,
144     ext: '.sqlite'
145 });
146 console.log(`db_path: ${db_path}`);
147
148 const csv_path = path.format({
149     dir: WORKCSV,
150     name: db_file,
151     ext: '.csv'
152 });
153 console.log(`csv_path: ${csv_path}`);
154
155 /*---DELETE--*/
156 if (options.hasOwnProperty('delete')) {
157
158     const WORKBAK_DB = path.format({
159         dir: WORKBAK,
160         name: 'db'
161     });
162     if (!fs.existsSync(WORKBAK_DB)) {
163         fs.mkdirSync(WORKBAK_DB, {recursive: true});
164     }
165
166     const WORKBAK_CSV= path.format({
167         dir:   WORKBAK,
168         name: 'csv'
169     });
170     if (!fs.existsSync(WORKBAK_CSV)) {
171         fs.mkdirSync(WORKBAK_CSV, {recursive: true});
172     }
173
174     const WORKBAK_LEDGER = path.format({
175         dir:   WORKBAK,
176         name: 'ledger'
177     });
178     if (!fs.existsSync(WORKBAK_LEDGER)) {
179         fs.mkdirSync(WORKBAK_LEDGER, {recursive: true});
180     }
181
182     // Backup workfin.sqlite, workfin.csv
183     const db_path_bak = path.format({
184         dir: WORKBAK_DB,
185         name: db_file,
186         ext: `.sqlite.${Date.now()}`
```

```
187     });
188
189     const csv_path_bak = path.format({
190         dir: WORKBAK_CSV,
191         name: db_file,
192         ext: `.csv.${Date.now()}`
193     });
194
195     try {
196         fs.renameSync(db_path, db_path_bak);
197         console.error(`Renamed ${db_path} to ${db_path_bak}`);
198         fs.renameSync(csv_path, csv_path_bak);
199         console.error(`Renamed ${csv_path} to ${csv_path_bak}`);
200     } catch (err) {
201         if (err.code === 'ENOENT')
202             console.log(`file ${db_path} and/or ${csv_path} did not exist; ignori
203         else {
204             throw err;
205         }
206     }
207
208     // Backup all .csv files
209     try {
210         const files = fs.readdirSync(WORKCSV);
211         files.forEach(file => {
212             const db_csv_path_file = path.format({
213                 dir: WORKCSV,
214                 name: file
215             });
216             const db_csv_path_bak  = path.format({
217                 dir: WORKBAK_CSV,
218                 name: file,
219                 ext: `.${Date.now()}`
220             });
221             fs.renameSync(db_csv_path_file, db_csv_path_bak);
222             console.log(`Renamed ${db_csv_path_file} to ${db_csv_path_bak}`);
223         });
224
225     } catch (err) {
226         if (err.code === 'ENOENT') {
227             console.log(`${db_csv_path} probably does not exist`);
228         } else {
229             throw err;
230         }
231     }
232
233     /* Ledger */
```

```
234     try {
235         const files = fs.readdirSync(WORKLEDGER);
236         files.forEach(file => {
237             if (!/zero/.test(file)) { // don't backup the zero ledger file
238                 const ledger_file = path.format({
239                     dir: WORKLEDGER,
240                     name: file
241                 });
242                 const ledger_file_bak = path.format({
243                     dir: WORKBAK_LEDGER,
244                     name: file,
245                     ext: `.${Date.now()}`
246                 });
247                 fs.renameSync(ledger_file, ledger_file_bak);
248                 console.log(`Renamed ${ledger_file} to ${ledger_file_bak}`);
249             }
250         });
251 
252     } catch (err) {
253         if (err.code === 'ENOENT') {
254             console.log(`${ledger_path} probably does not exist`);
255         } else {
256             throw err;
257         }
258     }
259 
260     process.exit(0);
261 }
262 
263 /*--ATTACH--*/
264 // attach in all situations except --delete
265 console.log('attaching...');
266 const db = new sqlite3.Database(db_path, err => {
267     if (err) {
268         return console.error(`Error opening database file ${db_path}: ${err.messa
269     }
270     console.log(`Successfully attached to database file ${db_path}`);
271 });
272 db.serialize();
273 db.run(`CREATE TABLE IF NOT EXISTS
274     usb (
275             rowid     INTEGER PRIMARY KEY NOT NULL,
276             acct      TEXT NOT NULL,
277             date      TEXT NOT NULL,
278             trans     TEXT NOT NULL,
279             checkno   TEXT,
280             txfr      TEXT,
```

```
281             payee     TEXT NOT NULL,
282             category  TEXT,
283             note      TEXT,
284             desc1     TEXT,
285             desc2     TEXT,
286             caseno    TEXT,
287             amount    REAL NOT NULL,
288             OrigPayee TEXT NOT NULL,
289             OrigMemo  TEXT NOT NULL )`);
290
291  // year is needed for checks, so define it here
292  const [acct,year] = options.csv;
293  if (!year) {
294    console.error('ERROR: year is not defined.');
295    process.exit(1);
296  }
297  const wl_year = parseInt(year, 10);
```

6 Find and Store Checks

After the Sqlite3 database is exported and converted by Ledger, there are numerous individual entries that need to be converted but for which there is no real data available to help. An example would be checks. The check information is located in the worklog, so one solution is to parse the worklog to obtain check information, then parse the Ledger file to update it.

I created a package called '`wlparser`' that can be used to find particular elements in the `worklog.<year>.otl` files. Among other things, this package is set up to find and return all checks for a particular year. At this point, the `findChecks` program requires an argument for a '`<year>`', and will then gather all checks and place them into the SQLite3 database in a table called '`checks`'. It can be run any number of times for any yeaer and will not add duplicate entries.

When this `csv-sqlite3` program converts the CSV files into the '`ledger`' format, it can do a search for a particular check number in this SQLite3 '`checks`' table and incorporate that data into the ledger file.

Use the commandline program `find-checks <year>` to find and store checks from the worklog yearly file identified by '`<year>`' option (i.e., `worklog.<year>.otl`) and save it into the SQLite3 database '`<db>`' in a table called '`checks`'.

```
 1  /* find-checks.js */
 2
 3  /* USAGE:
 4   * find-checks <year> [db]
 5   */
 6
 7  const DEFAULT_DB = 'workfin';
 8  const TABLE      = 'checks';
 9  const EXT        = '.sqlite';
10
11  // make sure WORKDB is defined
12  if (typeof process.env.WORKDB === 'undefined') {
13    console.error('Must defined environment variable for WORKDB');
14    process.exit(1);
15  }
16  const WORKDB = process.env.WORKDB;
17
18  // make sure a <year> argument is included
19  if (process.argv.length < 3) {
20    console.error('Must include a <year> argument: "find-checks <year>"');
21    process.exit(1);
22  }
23
24  // make sure the <year> argument is a number
25  const wlyear = parseInt(process.argv[2],10);
26  if (isNaN(wlyear)) {
27    console.error(`The <year> argument: "${process.argv[2]}" must be a year, e.g.,
28    process.exit(1);
29  }
30
31  // second optional argument is the name of the database, without extension
32  // if no second argument, use default db of $WORKDB/workfin.sqlite
33  const path   = require('path');
34  const db_path = path.format({
35    dir: WORKDB,
36    name: `${process.argv[3] || DEFAULT_DB}`,
37    ext: EXT
38  });
39
40  // Everything is a go; load the wlparser, wlchecks, sqlite3 modules
41  const {WLChecks} = require('wlparser');
42  const wlchecks   = new WLChecks(wlyear);
43  const sqlite3    = require('sqlite3').verbose(); // remove verbose() for producti
44  const CHECKS_COLS = [
45      'acct',
46      'checkno',
47      'date',
48      'payee',
49      'subject',
50      'purpose',
51      'caseno',
52      'amount'
```

Create Symbolic Link into WORKBIN

This code creates a symbolic link into the '`WORKBIN`' directory for the command line program '`find-checks`' so that it can be run as a script. Note that the link must be symbolic in order for the `node_modules/wlparser/lib/*` files to be found when `find-checks` is run from the '`WORKBIN`' directory.

```
ln -sf $PWD/find-checks.js $WORKBIN/find-checks
```

7 Export SQLite DB Data to CSV File

Once data has been downloaded from the bank's web site and imported into the sqlite3 database, it must be converted into a Ledger file. This is accomplished using the `--export` option.

7.1 '`--export`' Option

To export the SQLite3 database data to CSV and Ledger files, use the `--export` option:

```
--export [<csv-file>=workfin.csv] --csv <acct> <year>
```

The csv filename is optional, with the default being the same as the db file (i.e., `workfin`), with the extension `.csv`, (i.e., `workfin.csv`) in the `WORKCSV` directory. In order to facilitate matching an account to reconcile against, the `--export` option must be accompanied by the designation of a bank account, e.g., '`6815`' and a year, e.g., '`2016`'. These will filter the data that is exported from the `sqlite` data file and will become the account to reconcile against.

The export is accomplished by executing in a child process the command line program:

```
sqlite3 -header -csv db sql-statements
```

The child process runs the `sqlite3` command, and connects the '`STDOUT`' stream to the target CSV file. Since the entire database contents for a given year and account are exported, the output will truncate the target CSV file upon opening it for writing. The program will halt after the export.

7.2 Ledger `convert` Command

Upon an export of the SQLite3 data to a CSV file, the program will also send the exported data through the Ledger `convert` command and into the `workfin.ledger` data file.[1] This file is located in the `ledger/` directory below the `workfin` directory.

The `convert` command uses a `ledger` file filled with `ledger` '`directives`' to associate '`payee`' 's with '`account`' 's. If this '`directives`' file does not exist, then it will be created.

7.3 Export Code

```
115
116  /*--EXPORT--*/
117  if (options.hasOwnProperty('export')) {
118      const { spawnSync } = require('child_process');
119      const export_csv = options['export'] || db_file;
120      console.log(`exporting to ${export_csv}...`);
121
122      const export_csv_dir = WORKCSV;
123      if (!fs.existsSync(export_csv_dir)) {
124          fs.mkdirSync(export_csv_dir);
125          console.log(`Created ${export_csv_dir}`);
126      }
```

[1] Refer to the Ledger manual at Sec. 7.2.1.2 for the {{{command(convert}}}} command.

```
127     const export_csv_path = path.format({
128         dir: export_csv_dir,
129         name: export_csv,
130         ext: '.csv'
131     });
132
133     // --export must be accompanied by --csv <acct> <year> of the proper values
134     if (!options.hasOwnProperty('csv')) {
135         console.error('Export must be accompanied by a bank account (e.g., 6815),
136         process.exit(1);
137     }
138     const _acct = options.csv[0],
139           _year = options.csv[1];
140
141     if (!(Object.keys(DB_ACCTS).includes(_acct) && DB_YEARS.includes(_year))) {
142         console.error(`Invalid values for acct: ${_acct} or year: ${_year}`);
143         process.exit(1);
144     }
145
146     const usb_acct = `usb_${_acct}`;
147
148     //'as' - Open file for appending in synchronous mode. The file is created if
149     let fd = fs.openSync(export_csv_path,'as');
150     const size = fs.statSync(export_csv_path).size;
151     const header = size === 0 ? 'header' : 'noheader';
152     console.log(`export_csv_path: ${export_csv_path}`);
153
154     const sql = `
155 SELECT ${EXPORT_DB_COLS.join(',')}
156 FROM   usb
157 WHERE  acct = '${usb_acct}' and date like '${_year}%';`;
158
159     console.log(`sql: ${sql}`);
160
161     let ret = spawnSync(
162         'sqlite3',
163         [
164             db_path,
165             '-csv',
166             `-${header}`,
167             sql,
168         ],
169         {
170             encoding: 'utf-8',
171             stdio: [0,fd,2]
172         }
173     );
```

```
174
175     if (ret.error) {
176         console.log(`status: ${ret.status}\tsignal: ${ret.signal}`);
177         console.log(`error: ${ret.error}`);
178     }
179
180     console.log('done exporting');
181     fs.closeSync(fd);
182
183
184     /* CONVERT CSV TO LEDGER */
185     const ledger_dir = WORKLEDGER;
186     const ledger_path = path.format({
187         dir: ledger_dir,
188         name: export_csv,
189         ext: '.exported.ledger'
190     });
191     const zero_file = path.format({
192         dir: ledger_dir,
193         name: 'zero',
194         ext: '.ledger'
195     });
196     if (!fs.existsSync(ledger_dir)) {
197         fs.mkdirSync(ledger_dir);
198     }
199
200     //const l_file = fs.existsSync(ledger_path) ? ledger_path : zero_file;
201     const l_file = zero_file;
202
203     console.log(`converting: ${export_csv_path} to ledger_path: ${ledger_path}`);
204
205     fd = fs.openSync(ledger_path, 'as');        // 'as' - Open file for appending
206                                                 // The file is created if it does
207     ret = spawnSync(
208         'ledger',
209         [
210             'convert',
211             `${export_csv_path}`,
212             '--invert',
213             '--input-date-format=%Y-%m-%d',
214             `--account=Assets:${DB_ACCTS[_acct]}`,
215             '--rich-data',
216             `--file=${l_file}`,
217             `--now=${(new Date()).toISOString().split('T')[0]}`,
218         ],
219         {
220             encoding: 'utf-8',
```

```
221             stdio: [0,fd,2],
222         }
223     );
224
225     if (ret.error) {
226         console.log(`status: ${ret.status}\tsignal: ${ret.signal}`);
227         console.log(`error: ${ret.error}`);
228     }
229
230     fs.closeSync(fd);
231     process.exit(0);
232 }
233
234 /*--DON'T CONTINUE UNLESS --csv OPTION USED--*/
235 if (!options.hasOwnProperty('csv'))
236     process.exit(0);
```

7.4 The Zero Ledger File

The Zero Ledger File is a `ledger` file with an opening balance, list of accounts, and directives that associate '`payee`' 's with '`account`' 's. It is used by the `ledger convert` command to prepare a `ledger` file, create initial set of accounts, and parse a CSV file into the `ledger` format.

List of Accounts

1. Expenses—where money goes
2. Assets—where money sits
3. Income—where money comes from
4. Liabilities—where money is owed
5. Equity—where value is

Beneath theses top levels, there can be any level of detail required.

Allowable Accounts

Here are defined some allowable accounts:

```
account Expenses
account Assets
account Income
account Liabilities
account Equity
```

Use the `--strict` option to show incampatible accounts

Opening Balances

The first entry is a set of opening balances. It will look like this:

```
2016/01/01 * Opening Balance
    Expenses                                $0
        Assets:USB:Personal 6151             $0
        Assets:USB:Business 6815             $0
    Assets:USB:Trust 6831                     $0
    Assets:USB:Savings                      $0
    Income                                      $0
    Liabilities                              $0
        Equity:Opening Balance
```

Directives and Subdirectives

The Zero file uses two directives, each of which uses a sub-directive, of the form:

```
payee <PAYEE>
  alias </PAYEE_REGEX/>
account <FULL:ACCOUNT>
  payee </PAYEE_REGEX/>
```

In the above, the first line rewrites the '`payee`' field to establish a legitimate payee. The '`alias`' is a regex; anything that matches this directive will be turned into the associated '`payee`'. The second line uses an account and a '`payee`' directive to specify the proper '`account`'. Anything that matches the '`payee`' regex will be assigned the account.

```
payee USPS
 alias usps
account Expenses:Office:Postage
 payee ^(USPS)$

payee Staples
 alias staples
payee Ikea
 alias ikea
payee Portland Art Museum
 alias portland art
payee The Energy Bar
 alias energy bar
account Expenses:Office:Supplies
 payee ^(Staples|Portland Art Museum|Ikea|The Energy Bar)$

payee City of Portland
 alias city of portland
account Expenses:Business:Travel
 payee ^(City of Portland)$

payee RingCentral
```

```
 alias ringcentral
payee AT&T
 alias (at&?t)
payee CenturyLink
 alias (centurylink|ctl)
payee NameBright
 alias namebright
account Expenses:Business:Communication
 payee ^(RingCentral|AT\&T|CenturyLink|NameBright)$

payee AVVO
 alias avvo
account Expenses:Business:Advertising
 payee AVVO

payee National Law Foundation
 alias national law fou
payee Coursera
 alias coursera
payee EdX Inc.
 alias edx
account Expenses:Professional:CLE
 payee ^(National Law Foundation|Coursera|EdX Inc.)$

payee State Bar of CA
 alias state bar of ca
account Expenses:Professional:License
 payee ^(State Bar of CA)$

payee Costco Gas
 alias costco gas
account Expenses:Office:Transportation
 payee ^(Costco Gas)$

payee American Express
 alias amex
payee Citi
 alias citi
account CC:Payment
 payee ^(American Express|Citi)$

payee Apple Store
 alias apple
payee Radio Shack
 alias radioshack
account Expenses:Office:Supplies
 payee ^(Apple Store|Radio Shack)$
```

```
payee State Bar WA
 alias interest paid this period
account Trust:LTAB
 payee State Bar WA

payee State Bar WA
 alias ltab
account Trust:LTAB
 payee State Bar WA

account Assets:Personal
 payee usb_6151
```

{{{heading(An '`include`' File }}}

Finally, include a file with an '`Accounts:Payable`' Category:

```
include accounts_payable.ledger
```

7.5 The Accounts Payable File

The `accounts_payable.ledger` file contains any outstanding accounts that should be included to make the inputted data correct, such as a set of outstanding invoices:

```
2016/01/18 * Clark County Indigent Defense ; Invoice No.s 092--099
        Assets:Accounts Receivable        $1852.50
        Income:120703                        -$  82.50 ; Invoice No 092
        Income:140707                        -$ 525.00 ; Invoice No 093
        Income:140709                        -$ 397.50 ; Invoice No 094
        Income:150701                        -$  15.00 ; Invoice No 095
        Income:150704                        -$ 742.50 ; Invoice No 096
        Income:150705                        -$   7.50 ; Invoice No 097
        Income:150706                        -$   7.50 ; Invoice No 098
        Income:150707                        -$  75.00 ; Invoice No.099
```

8 Process CSV Files

The Node.js module `csv` (`https://csv.js.org/`) contains the

- csv-parser (`https://csv.js.org/parse/`),
- csv-stream-transformer (`https://csv.js.org/transform/`),
- csv-stringifier (`https://csv.js.org/stringify/`),

all of which will be used in this project. The pattern is to open a CSV file, parse a CSV string into records and pipe those records through the transformer to be massaged into shape. From there the new data is saved in another CSV file and also an SQLite3 database using

- `sqlite3` (`https://www.npmjs.com/package/sqlite3`) (see its API (`https://github.com/mapbox/node-sqlite3/wiki/API`) also)

The processing of a CSV file, therefore, involves the following steps and Node.js modules:

1. Find the correct CSV file (using `FileSystem`) and open it as a Readable Stream (`https://nodejs.org/dist/latest-v12.x/docs/api/stream.html#stream_readable_streams`);
   - Section 8.4 [Set Up StreamReader], page 43,
2. Open a new CSV file to hold the new transformed data as a Writable Stream (`https://nodejs.org/dist/latest-v12.x/docs/api/stream.html#stream_writable_streams`)
   - Section 8.1 [Set Up CSV-Stringify], page 36,
3. Open an SQLite3 database to hold the new transformed data
   - Chapter 5 [Attach To or Delete a Database], page 21,
4. Read the CSV records from the file as a string (using `StreamReader`)
   - Section 8.4 [Set Up StreamReader], page 43,
5. Parse the string into JS records (using `CSV-Parse`)
   - Section 8.3 [Set Up CSV-Parse], page 43,
6. Transform the JS records into usable data (using `CSV-Transform`)
   - Section 8.2.2 [Set Up the Stream Transform], page 42,
   - Section 8.2.1 [Set Up the Transform Function], page 38,
7. Save the new data in the new CSV file (using `StreamWriter`)
   - Section 8.1 [Set Up CSV-Stringify], page 36,
8. Save the new data in an SQLite3 database (using `SQLite3`)
   - Section 8.2.2 [Set Up the Stream Transform], page 42,

8.1 Set Up CSV-Stringify

This section receives the transformed records from the Transform function and writes them to new csv files. The new csv files will be located in the '`WORKCSV`' directory, e.g., `WORK/workfin/csv`. A file will be called, for example, `usb_6815__2016.csv`. Notice that

this file name uses two underscores, whereas the source files use two dashes; in all other respects, they are the same.

```
237 const stringifier = csv.stringify({
238     header: true,
239     columns: DB_COLS,
240 });
241
242 const usb_acct = `usb_${acct}`;
243 const usb_acct_year = `${usb_acct}__${year}.csv`;
244
245 const csv_path_file = path.join(WORKCSV, usb_acct_year);
246 console.log(`CSV PATH FILE: ${csv_path_file}`);
247
248 let csv_stringifier;
249 try {
250     csv_stringifier = fs.createWriteStream(csv_path_file);
251     console.log(`WRITE STREAM: ${csv_path_file} has been successfully opened.`);
252 } catch (err) {
253     console.error(err.message);
254     process.exit(1);
255 }
256
257 stringifier.on('readable', function() {
258     console.log('stringifier is now readable');
259     let row;
260     while (row = this.read()) {
261         console.log(`stringifer row: ${row}`);
262         csv_stringifier.write(row);
263     }
264 });
265
266 stringifier.on('error', function(err) {
267     console.error(err.message);
268 });
269
270 stringifier.on('finish', function() {
271     console.log('stringifier is done writing to csv_stringifer');
272     csv_stringifier.end('stringifer called csv_stringifier\'s "end" method');
273 });
274
275 stringifier.on('close', function() {
276     console.log('stringifier is now closed');
277 });
278
279 csv_stringifier.on('close', function() {
280     console.log('csv_stringifier is now closed');
```

```
281  });
```

8.2 Set Up Stream-Transform and Transform Function

This code implements the stream transformer functionality, which is at the heart of this project.

The Transformer is a Node.js Transform Stream (`https://nodejs.org/dist/latest-v12.x/docs/api/stream.html#stream_class_stream_transform`). This means it is capable of both reading and writing data. In this project, it [writes data], page 43, in the CSV Parser, and then reads it here via its `transformer.read()` method. This `transformer` object has a `transform()` method that takes a function callback, whose purpose is to to *transform* records that are read. This is the heart of this project. The `transform()` function is implemented in the following section, and returns completely transformed CSV bank records at its end. These transformed records are then written to both a new CSV file, and the SQLite3 database.

`transform` (*transform_callback*) [Method on `CSV`]

The CSV `transform` method reads a record and sends that record to a '`TRANSFORM_CALLBACK`' that is used to *transform* the data.

After it transforms the data, the transformer receives the new data via a '`readable`' event, where it can process the data.

The transformed data will also be saved into the SQLite3 database via an '`INSERT`' statement executed by the `db.run()` method.

8.2.1 Set Up the Transform Function

The Transform Function receives a record and massages it into shape. The following regular expressions were created based upon inspection of the raw data as it came from the bank for years 2016, 2017, and 2018. It does a decent job of creating readable payees and memos, as well as txfrs (transfers), but it has not been set up to do anything for check payees, categories or related records, for example.

```
282  const transform_function = function (record) {
283      const DEBIT   = 'debit';
284      const CREDIT  = 'credit';
285      const CHECK   = 'check';
286      const CASH    = 'cash';
287      const DEPOSIT = 'deposit';
288      const UNKNOWN = 'unknown';
289      const TRANS   = 'transfer';
290      const USBANK  = 'usbank';
291      let   trfrom  = '';
292
293      // Add new columns: acct, checkno, txfr, caseno, desc1, desc2, category
294      record.acct    = usb_acct;
295      record.checkno = null; // check no.
296      record.txfr    = null; // direction and acct #
297      record.caseno  = null; // related case foreign key
```

```
298     record.desc1   = null; // noun
299     record.desc2   = null; // adjective
300     record.category= null; // categorization of the transaction
301
302     // Format date as yyyy-mm-dd; delete original Date
303     record.date = new Date(record['Date']).toISOString().split('T')[0];
304     delete record['Date'];
305
306     // Change Transaction to trans; delete original Transaction
307     record.trans = record['Transaction'].toLowerCase();
308     delete record['Transaction'];
309
310     // Change Amount to amount as Currency type; delete original Amount
311     record.amount = accounting.formatMoney(record['Amount']);
312     delete record['Amount'];
313
314     // Change Name to payee; keep original Name as OrigName; delete Name
315     record.payee = record['Name'].toLowerCase().trimRight();
316     record.OrigPayee = record['Name'];
317     delete record['Name'];
318
319     // Clean up Memo by removing Download message; return as note; keep Memo as O
320     let re = new RegExp('Download from usbank.com.\\s*');
321     record.note = record['Memo'].replace(re,'').toLowerCase();
322     record.OrigMemo = record['Memo'];
323     delete record['Memo'];
324
325     // Add check no. to checkno column
326     if (record.payee === CHECK) {
327         const checkno = record.trans.replace(/^0*/,'');
328         record.checkno  = checkno;
329         record.trans   = DEBIT;
330         record.payee  = `(${record.checkno}) check`;
331         record.note  += `Purchase by check no. ${checkno}`;
332         record.desc1  = 'purchase';
333         record.desc2  = 'check';
334     }
335
336     if (record.payee.match(/(returned) (item)/)) {
337         record.desc1 = RegExp.$2;
338         record.desc2 = RegExp.$1;
339         record.payee = USBANK;
340         record.note = `${record.desc2} ${record.desc1}`;
341     }
342
343     if (record.payee.match(/(internet|mobile) (banking) transfer (deposit|withdra
344         record.desc1 = RegExp.$3;
```

```
345         record.desc2 = RegExp.$1;
346         record.txfr = `${(RegExp.$3 === 'deposit') ? '<' : '>'} usb_${RegExp.$4}`
347         tofrom = (record.trans === 'debit') ? 'to' : 'from';
348         record.payee = (record.trans === 'debit') ? `usb_${RegExp.$4}` : `usb_${o
349         record.note = `${record.desc2} ${record.desc1}: ${TRANS} ${tofrom} ${reco
350         if (/>/.test(record.txfr)) {
351             record.payee = `Transfer to ${record.payee} from ${record.acct}`;
352         } else {
353             record.payee = `Transfer to ${record.payee} from usb_${RegExp.$4}`;
354         }
355     }
356
357     if (record.payee.match(/debit (purchase)\s*-?\s*(visa)? /)) {
358         record.desc1 = RegExp.$1;
359         record.desc2 = RegExp.$2;
360         record.payee = record.payee.replace(RegExp.lastMatch,'');
361         record.note = `${record.desc2} ${record.desc1} ${record.note}`.trimLeft()
362     }
363
364     // Removed ELECTRONIC WITHDRAWAL for payment to State Bar of CA
365     if (record.payee.match(/^.*(state bar of ca)/)) {
366         record.payee = RegExp.$1;
367     }
368
369     // web authorized payment
370     // atm|electronic|mobile check|rdc deposit|withdrawal <name>
371     if (record.payee.match(/(web authorized) (pmt) |(atm|electronic|mobile)?\s*(c
372         tofrom = '';
373         record.desc1 = RegExp.$2 ? RegExp.$2 : RegExp.$4 ? RegExp.$4 : RegExp.$5
374         record.desc2 = RegExp.$1 ? RegExp.$1 : RegExp.$3 ? RegExp.$3 : 'undefined
375         if (RegExp.$3 === 'atm' || RegExp.$3 === 'electronic' || RegExp.$3 === 'm
376             record.payee = (RegExp.$5 === 'deposit') ? `usb_${options.csv[0]}` :
377         } else {
378             record.payee = record.payee.replace(RegExp.lastMatch,'');
379         }
380         if (record.note.match(/paypal/) && record.trans === CREDIT) {
381             record.txfr = `< ${RegExp.lastMatch}`;
382             tofrom = ' from';
383         }
384         record.note = `${record.desc2} ${record.desc1}${tofrom} ${record.note}`.t
385     }
386
387     if (record.payee.match(/(zelle instant) (pmt) (from (\w+\s\w+))\s(.*)$/)) {
388         record.desc1 = RegExp.$2;
389         record.desc2 = RegExp.$1;
390         record.note = `${record.desc2} ${record.desc1} ${RegExp.$3}`;
391         record.payee = `usb_${options.csv[0]}`;
```

```
392     }
393
394     if (record.payee.match(/(overdraft|international) (paid|processing) (fee)/)) {
395         record.desc1 = RegExp.$3;
396         record.desc2 = `${RegExp.$1} ${RegExp.$2}`;
397         record.payee = USBANK;
398         record.note  = `${record.desc2} ${record.desc1} to ${record.payee}`;
399     }
400
401     record.payee = record.payee.replace(/\s*portland\s{2,}or$|\s*vancouver\s{2,}w
402     record.note  = record.note.replace(/\s*portland\s{2,}or$|\s*vancouver\s{2,}wa
403     record.payee = record.payee.replace(/\s\d{3}\w+\s{2,}or$/,''); // Nike Compan
404     record.note  = record.note.replace(/\s\d{3}\w+\s{2,}or$/,'');
405     record.payee = record.payee.replace(/\s*[-\d]{5,}\s*\w{2}$/,''); // '650-4724
406     record.note  = record.note.replace(/\s*[-\d]{5,}\s*\w{2}$/,'');
407     record.payee = record.payee.replace(/(\s\w*https)?www.*$/,''); // WWW.ATT.COM
408     record.note  = record.note.replace(/(\s\w*https)?www.*$/,'');
409     record.payee = record.payee.replace(/\s*\w+\.com\s+\w{2}$/, '');
410     record.note  = record.note.replace( /\s*\w+\.com\s+\w{2}$/, '');
411     record.payee = record.payee.replace(/aws.amazon.cWA/i,''); // serviaws.amazon
412     record.note  = record.note.replace(/aws.amazon.cWA/i,'');
413     if (record.payee.match(/(bostype \/ wes bo)(hamilton\s+on)/)) { // WES BOHAMI
414         record.payee = 'Wes Bos';
415         record.note  = record.note.replace(RegExp.$1,'Wes Bos');
416         record.note  = record.note.replace(RegExp.$2, '');
417     }
418     record.payee = record.payee.replace(/\s{2,}/g,' ');
419     record.note  = record.note.replace(/\s{2,}/g,' ');
420
421     /*
422       'DEBIT PURCHASE -VISA SQ *PHIL        877-417-4551WA'
423
424       You paid Phil $159 for Atreus keyboard kit and shipping
425
426       It is for a credit card processor that goes by the brand name
427       Square Up. Merchants can run credit card transactions through
428       their iPhone or iPads using the Square Up services. Mine was for
429       a taxi ride. https://800notes.com/Phone.aspx/1-877-417-4551
430     */
431
432     record.payee = record.payee.replace(/sq/, 'square');
433     record.note  = record.note.replace(/sq/, 'square');
434
435     return record;
436 }
```

8.2.2 Set Up the Stream Transform

```
437 const transformer = csv.transform(transform_function);
438
439 /* TRANSFORMER reads records through its TRANSFORM_FUNCTION */
440 /* -------------------------------------------------------- */
441 transformer.on('readable', function() {
442     let record;
443     while ((record = transformer.read())) {
444         console.log(`Transformer record:\n${util.inspect(record)}`);
445
446         /* STRINGIFIER WRITE Records */
447         /* ------------------------- */
448         stringifier.write(record);
449
450
451
452         /* DB RUN---INSERT RECORDS */
453         /* ----------------------- */
454         const tab_name  = DB_TABLES['usb'];
455         const col_names = DB_COLS.join(',');
456         const col_phs   = DB_COLS.map(c => '?').join(',');
457         const col_values= DB_COLS.map(c => record[c]);
458
459         let sql = `INSERT INTO ${ tab_name }( ${ col_names } )
460                    VALUES ( ${ col_phs } )`;
461
462         console.log(`sql: ${ sql }`);
463         console.log(`col_values: ${ col_values }`);
464
465         db.run(sql, col_values, (err) => {
466            if (err) {
467                console.error(err.message);
468                console.error(`ERROR sql: ${ sql }`);
469                console.error(`ERROR values: ${ col_values }`);
470                process.exit(1);
471            }
472        });
473    }
474 });
475
476 transformer.on('error', function(err) {
477     console.error(err.message);
478 });
479
480 transformer.on('finish', function() {
481     console.log('Transformer finished writing records.');
```

```
482 });
483
484 transformer.on('end', function() {
485     console.log('Transformer end reached.');
486     stringifier.end();
487 });
```

8.3 Set Up CSV-Parse

This section implements the csv parser. By default, it does little other than read a large string of data and parse it into an array of records. By giving it the option '`columns = true`', however, the parser will use the first line as a list of column headings, and produce an array of objects where the keys are column names, and the values are column entries. Each record is written to the stream transformer using its '`WRITE`' method.

```
488 const parser = csv.parse({columns: true});
489 const records = [];
490
491 parser.on('readable', function() {
492     console.log('Parser beginning to read records.');
493     let record;
494
495     /* PARSE A RECORD AND WRITE TO THE TRANSFORMER */
496     while ((record = parser.read())) {
497         console.log(`parser record:\n${util.inspect(record)}`);
498         transformer.write(record);
499     }
500
501 });
502
503 parser.on('error', function(err) {
504     console.error(err.message);
505 });
506
507 parser.on('end', function() {
508     console.log('Parser finished reading records.');
509 });
510
511 parser.on('finish', function () {
512     console.log('Parser finished writing records.');
513     console.log('Parser calling transformer end');
514     transformer.end();
515 });
```

8.4 Set Up StreamReader

This section implements the Stream Reader that reads the CSV file in as a large string of data and sends it to the csv parser via the parser's `write` method.

CSV financial files are found in the directories '`$WORKUSB_[6815|6831]/yyyy`', where '`yyyy`' can be 2004–2019, and on. Given '`[6815|6831]`' and a year '`[2004|2005...2019]`', the file path will be '`$WORKUSB_6815/YYYY/usb_6815--yyyy.csv`'. This code makes sure the file exists and the user has proper permissions to read it before proceeding.

```
516  if (options.csv) {
517      // const acct = options.csv[0],
518      //       year = options.csv[1];
519      // const usb_acct = `usb_${acct}`;
520
521      if (!process.env.WORKUSB) {
522          console.error('You must assign a path to the shell variable WORKUSB');
523          process.exit(1);
524      }
525
526      const acct_year_path = `${process.env.WORKUSB}/${usb_acct}/${year}`;
527      const acct_year_csv_file = `${usb_acct}--${year}.csv`;
528      const acct_year_csv_file_path = `${acct_year_path}/${acct_year_csv_file}`;
529      if (!fs.existsSync(acct_year_csv_file_path) || !(fs.accessSync(acct_year_csv_
530          console.error(`Cannot find or access the CSV file at '${acct_year_csv_fil
531          process.exit(1);
532      }
533      console.log(`Successfully found the CSV file: '${acct_year_csv_file_path}'`);
534
535      /* CREATE THE STREAM HERE */
536      const csv_file_stream = fs.createReadStream(acct_year_csv_file_path, {encodin
537
538      /* Set up streaming events 'READABLE', 'ERROR', and 'END' */
539      csv_file_stream.on('readable', function () {
540          let record;
541
542          /* READ THE RECORDS */
543          while ((record = this.read())) {
544              console.log(`readable record: ${record}`);
545
546              /* WRITE A RECORD TO THE PARSER */
547              parser.write(record);
548
549          }
550          parser.end();
551
552      });
553
554      csv_file_stream.on('error', function(err) {
555          console.error(err.message);
556      });
557
```

```
558      csv_file_stream.on('end', function () {
559          console.log('Reader finished reading data.');
560      });
561  }
```

Appendix A Node-SQLite3 Module

Asynchronous, non-blocking SQLite3 bindings for Node.js.

- Github (`https://github.com/mapbox/node-sqlite3`)
- Wiki API (`https://github.com/mapbox/node-sqlite3/wiki/API`)

A.1 Node-SQLite3 Module Usage

```
1  var sqlite3 = require('sqlite3').verbose();
2  var db = new sqlite3.Database(':memory:');
3
4  db.serialize(function() {
5    db.run("CREATE TABLE lorem (info TEXT)");
6
7    var stmt = db.prepare("INSERT INTO lorem VALUES (?)");
8    for (var i = 0; i < 10; i++) {
9        stmt.run("Ipsum " + i);
10   }
11   stmt.finalize();
12
13   db.each("SELECT rowid AS id, info FROM lorem", function(err, row) {
14       console.log(row.id + ": " + row.info);
15   });
16 });
17
18 db.close();
```

A.2 Features

- Straightforward query and parameter binding interface
- Full Buffer/Blob support
- Extensive debugging support
- Query serialization API
- Extension support
- Big test suite
- Written in modern C++ and tested for memory leaks
- Bundles Sqlite3 3.26.0 as a fallback if the installing system doesn't include SQLite

A.3 Node-SQLite3 API

`node-sqlite3` has built-in *function call serialization* and automatically waits before executing a blocking action until no other action is pending. This means that it's safe to start calling functions on the database object even if it is not yet fully opened. The `Database#close()` function will wait until all pending queries are completed before closing the database.

A.4 Node-SQLite3 Control Flow—Two Execution Modes

`node-sqlite3` provides two functions to help control the execution flow of statements. The default mode is to execute statements in *parallel*. However, the `Database#close` method will always run in *exclusive mode*, meaning it waits until all previous queries have completed and `node-sqlite3` will not run any other queries while a `close` is pending.

A.4.1 Serialize Execution Mode

`serialize` ([*callback*]) [Method on `Database`]

Puts the *execution mode* into *serialized mode*. This means that at most one statement object can execute a query at a time. Other statements wait in a queue until the previous statements are executed.

If a callback is provided, it will be called immediately. All database queries scheduled in that callback will be serialized. After the function returns, the database is set back to its original mode again.

Calling `Database#serialize()` within nested functions is safe:

```
 1  // Queries scheduled here will run in parallel.
 2
 3  db.serialize(function() {
 4
 5      // Queries scheduled here will be serialized.
 6      db.serialize(function() {
 7          // Queries scheduled here will still be serialized.
 8      });
 9      // Queries scheduled here will still be serialized.
10  });
11
12  // Queries scheduled here will run in parallel again.
13
```

Note that queries scheduled not directly in the callback function are not necessarily serialized:

```
 1  db.serialize(function() {
 2
 3      // These two queries will run sequentially.
 4      db.run("CREATE TABLE foo (num)");
 5      db.run("INSERT INTO foo VALUES (?)", 1, function() {
 6
 7          // These queries will run in parallel and the second query will probably
 8          // fail because the table might not exist yet.
 9          db.run("CREATE TABLE bar (num)");
10          db.run("INSERT INTO bar VALUES (?)", 1);
11      });
12  });
```

If you call it without a function parameter, the execution mode setting is sticky and won't change until the next call to `Database#parallelize`.

A.4.2 Parallelize Execution Mode

`parallelize` ([*callback*]) [Method on `Database`]

Puts the execution mode into parallelized. This means that queries scheduled will be run in parallel.

If a callback is provided, it will be called immediately. All database queries scheduled in that callback will run parallelized. After the function returns, the database is set back to its original mode again.

Calling `Database#parallelize()` within nested functions is safe:

```
1  db.serialize(function() {
2
3      // Queries scheduled here will be serialized.
4      db.parallelize(function() {
5
6          // Queries scheduled here will run in parallel.
7      });
8
9      // Queries scheduled here will be serialized again.
10 });
```

If you call it without a function parameter, the execution mode setting is sticky and won't change until the next call to `Database#serialize`.

Appendix B Converting CSV Files

B.1 Ledger Convert Command

- Ledger manual sec. 7.2.1.2

The `convert` command parses a comma separated value (csv) file and prints Ledger transactions. Importing csv files is a lot of work, but is very amenable to scripting.

```
$ ledger convert download.csv --input-date-format "%m/%d/%Y"
```

Fields Descriptors

Your bank's csv files will have fields in different orders from other banks, so there must be a way to tell Ledger what to expect. Ledger expects the first line to contain a description of the fields on each line of the file.

- Insert a line at the beginning of the csv file that describes the fields to Ledger.
- The fields ledger can recognize contain these case-insensitive strings:
  - date
  - posted
  - code
  - payee
  - desc / description
  - amount
  - cost
  - total
  - note
- the '`Input Date Format`' option tells ledger how to interpret the dates:
  - '`--input-date-format DATE_FORMAT`'
  - e.g., '`"%m/%d/%Y"`'
- Metadata
  - If there are columns in the bank data you would like to keep in your ledger data, besides the primary fields described above, you can name them in the field descriptor list and Ledger will include them in the transaction as meta data if it doesn't recognize the field name.
  - CSV original line from the csv file
  - Imported date data imported
  - UUID unique checksum; if an entry with the same 'UUID' tag is already included in the normal ledger file (specified via `--file FILE (-f)` or via the environment variable '`LEDGER_FILE`') this entry will not be printed again.

Convert Command Options

The convert command accepts the following options:

–invert inverts the amount field

–auto-match
automatically matches an account from the Ledger journal for every CSV line

–account STR
use to specify the account to balance against

–rich-data stores additional tag/value pairs

–file (-f) the normal ledger file (could also be specified via the environment variable '`LEDGER_FILE`')

–input-date-format
tells ledger how to interpret the dates

Command Directives

You can also use `convert` with '`payee`' and '`account`' directives.

1. Directives

   a. '`directive`' :: Command directives must occur at the beginning of a line.

   b. '`account`' directive :: Pre-declare valid account names. This only has an effect if `--strict` or `--pedantic` is used. The '`account`' directive supports several optional sub-directives, if they immediately follow the '`account=`' directive and if they begin with whitespace:

```
account Expenses:Food
              note This account is all about the chicken!
              alias food
              payee ^(KFC|Popeyes)$
              check commodity == "$"
              assert commodity == "$"
              eval print("Hello!")
              default
```

   c. '`payee`' sub-directive :: The '`payee`' sub-directive of the '`account`' directive, which can occur multiple times, provides regexes that identify the '`account`' if that payee is encountered and an '`account`' within its transaction ends in the name "Unknown".

```
2012-02-27 KFC
              Expenses:Unknown      $10.00  ; Read now as "Expenses:Food"
              Assets:Cash
```

   d. '`alias`' sub-directive :: The '`alias`' sub-directive of the '`acount`' directive, which can occur multiple times, allows the alias to be used in place of the full account name anywhere that account names are allowed.

   e. '`payee`' directive :: The payee directive supports two optional sub-directives, if they immediately follow the payee directive and——if it is on a successive line——begins with white-space:

```
payee KFC
```

```
    alias KENTUCKY FRIED CHICKEN
    uuid 2a2e21d434356f886c84371eebac6e44f1337fda
```

The '`alias`' sub-directive of the '`payee`' directive provides a regex which, if it matches a parsed payee, the declared payee name is substituted:

f. '`alias`' directive :: Define an alias for an account name. The aliases are only in effect for transactions read in after the alias is defined and are affected by '`account`' directives that precede them. The '`alias`' sub-directive, which can occur multiple times, allows the alias to be used in place of the full account name anywhere that account names are allowed.

```
alias Dining=Expenses:Entertainment:Dining
alias Checking=Assets:Credit Union:Joint Checking Account

2011/11/28 YummyPalace
    Dining        $10.00
    Checking
```

2. First, you can use the '`payee`' directive and its '`alias`' sub-directive to rewrite the '`payee`' field based on some rules.

```
payee Aldi
      alias ^ALDI SUED SAGT DANKE
```

3. Then you can use the '`account`' directive and its '`payee`' sub-directive to specify the account.

```
account Aufwand:Einkauf:Lebensmittel
       payee ^(Aldi|Alnatura|Kaufland|REWE)$
```

Directive Example

```
payee Aldi
          alias ^ALDI SUED SAGT DANKE
account Aufwand:Einkauf:Lebensmittel
          payee ^(Aldi|Alnatura|Kaufland|REWE)$
```

Note that it may be necessary for the output of `ledger convert` to be passed through `ledger print` a second time if you want to match on the new '`payee`' field. During the `ledger convert` run, only the original '`payee`' name as specified in the csv data seems to be used.

9 Update package.json Version

In order to keep the `package.json` '`version`' number, Listing 3.1, in sync with this document's version number, I have created a little script to update its version based upon the macro '`version`''s current value. This macro is defined at the very top of this Org source file just below the title and date. I update this version number after every modification to this source file and before commiting the change. This little script will then be run whenever an installation occurs.

This script can be run by invoking the Makefile target '`update-version`'. This will checkout the '`dev`' branch, tangle the `update-version.sh` script into the `scripts/` directory, run it, amend the most recent commit to include the updated version number, push the amended commit to Github, and finally create a '`prod`' branch (for '`production`'), install all of the files and documentation, and commit and push the '`prod`' branch to Github. At this point, the package is ready to be cloned from Github and contains the most recent version number, the dependencies installed, and run.

This program is a little `sed)` script that modifies this Org source file in-place (after creating and storing a backup of the source) by copying the version number found in the macro and updating the version number of the `package.json` file. It runs very quickly.

```
sed -i .bak -E -e '
/\#\+macro: version Version/bx
/,\/\\"version\\":/by
b
:x
h
s/^(.*macro: version Version )(.*)$/\2/
x
b
:y
H
x
s/\n//
s/^([[:digit:]]+\.[[:digit:]]+\.[[:digit:]]+)(.*)([[:digit:]]+\.[[:digit:]]+\.[[:digit
' CSV-SQLite3.org
```

Listing 9.1: Update `package.json` Version Number

10 Makefile

```
SOURCE = CSV-SQLite3
ORG    = $(SOURCE).org
TEXI   = $(SOURCE).texi
INFO   = $(SOURCE).info
PDF    = $(SOURCE).pdf
DOCS   = docs
SCRIPTS=scripts

.PHONY: clean clean-world clean-prod
.PHONY: tangle weave texi info pdf
.PHONY: install install-docs install-info install-pdf open-pdf docs-dir
.PHONY: update-dev update-prod checkout-dev checkout-prod
.PHONY: update-version tangle-update-version run-update-version

texi: $(TEXI)
$(TEXI): $(ORG)
        emacs -Q --batch $(ORG) \
        --eval '(setq org-export-use-babel nil)' \
        --eval '(org-texinfo-export-to-texinfo)'

tangle: $(ORG)
        emacs -Q --batch $(ORG) \
        --eval '(org-babel-tangle-file "$(ORG)")'

info weave install-info: $(DOCS)/$(INFO)
$(DOCS)/$(INFO): $(TEXI) | docs-dir
        makeinfo --output=$(DOCS)/ $(TEXI)

install: package.json
package.json:        $(ORG) | docs-dir
        emacs -Q --batch $(ORG) \
        --eval '(require '\''ob-shell)' \
        --eval '(require '\''ob-js)' \
        --eval '(setq org-confirm-babel-evaluate nil)' \
        --eval '(org-texinfo-export-to-info)'
        mv $(INFO) $(DOCS)/
        make install-pdf

install-docs: install-info install-pdf

pdf install-pdf: $(DOCS)/$(PDF)
$(DOCS)/$(PDF): $(TEXI) | docs-dir
        pdftexi2dvi -q -c $(TEXI)
        mv $(PDF) $(DOCS)/
```

```
open-pdf: $(DOCS)/$(PDF)
	open $(DOCS)/$(PDF)

docs-dir: $(DOCS)
$(DOCS):
	mkdir -vp docs


update-version: update-dev update-prod

checkout-dev:
	git checkout dev

update-dev: checkout-dev run-update-version
	git add -u
	git commit --amend -C HEAD
	git push origin +dev

checkout-prod: clean-world checkout-dev
	git checkout -B prod

update-prod: checkout-prod install clean-prod
	git add -A .
	git commit -m "Branch:prod"
	git push origin +prod

run-update-version: tangle-update-version
	./$(SCRIPTS)/update-version.sh
	mv -v $(ORG).bak $(WORKBAK)/$(ORG).$(shell date "+%s")

tangle-update-version: $(SCRIPTS)/update-version.sh
$(SCRIPTS)/update-version.sh: $(ORG)
	emacs -Q --batch $(ORG) \
	--eval '(search-forward ":tangle scripts/update-version.sh")' \
	--eval '(org-babel-tangle '\''(4))'


clean:
	-rm *~

clean-world: clean
	-rm *.{texi,info,pdf,js,json,lock,log,bak}
	-rm -rf LogReader
	-rm -rf node_modules $(SCRIPTS) $(DOCS)

clean-prod: clean
	-rm *.{texi,org} Makefile LogReader
```

```
-rm -rf node_modules
```

Index

—

`--create` ... 19
`--delete` ... 19
`--export` option ... 29
`--file` option ... 50
`--help` ... 19
`--input-data-format` option ... 50

A

`account` ... 33
`account` directive ... 50
`account` option ... 50
accounts ... 32
`alias` directive ... 50
arguments ... 19
`auto-match` option ... 50

C

`cl_usage()` ... 19
columns, usb table ... 7
command directives ... 50
command-line arguments ... 19
command-line usage ... 19
`command-line-arguments` ... 19
`command-line-usage` ... 19
`convert` command ... 29, 32
`convert` command, Ledger ... 49
convert option, `--account` ... 50
convert option, `--auto-match` ... 50
convert option, `--file` ... 50
convert option, `--input-data-format` ... 50
convert option, `--rich-data` ... 50
convert option, `invert` ... 50
csv file fields ... 49
csv file, parse ... 49

D

database file `db` ... 21
`db` database file ... 21
`db.run` ... 38
directives ... 33, 50
directives, Ledger ... 29

E

environment variable *LEDGER_FILE* ... 50
exclusive mode ... 47
execution flow ... 47
execution mode, parallelized ... 48
execution mode, serialize ... 47
execution mode, sticky ... 47

F

fields, csv file ... 49

I

`INSERT` into `db` ... 38
`invert` option ... 50

L

Ledger `convert` command ... 49
LEDGER_FILE environment variable ... 50

M

metadata ... 49

O

opening entry ... 32

P

parallel execution ... 47
parallized exeuction mode ... 48
parse csv file ... 49
`payee` ... 33
`payee` directive ... 50

R

`rich-data` option ... 50

S

serialization ... 46
serialization, function call ... 46
serialize execution mode ... 47
serialized mode ... 47
sticky execution mode ... 47

T

tables ... 7

U

usage 19
usb table columns 7

V

verbose mode 21

W

`write` method, transformer 43

Function Index

P

`parallelize` on `Database` 48

S

`serialize` on `Database` 47

T

`transform` on `CSV` 38

Listings

Listing 1: Source code for catuniq.pl 4
Listing 1.1: Define the USB Table Schema 8
Listing 1.2: Define Checks Table Schema 10
Listing 3.1: Create the CSV-SQLite3 Project 12
Listing 3.2: Package json 16
Listing 6.1: The script file 27
Listing 9.1: Update `package.json` Version Number 52