Data curation with SQL

Manage your data the “enterprise” way

Robert Forkel

Quantitative Methods – Masterclass 2016

Max Planck Institute for the Science of Human History

Research Data Curation

In the following, we assume data curation scenarios within the following boundary conditions

- Structured data of the type that is often kept in (multiple) Excel sheets
- Possibly related multimedia objects (images, audio, video)
- Small to medium size, i.e. $< 1,000,000$ rows in tables, $< 100GB$ of data on disc
- Mainly curated by hand by a small team

Research Data Examples

Examples for this kind of research data:

- The data served by the D-PLACE app
- WALS - the World Atlas of Language Structures
- Tsammalex - a multilingual lexical database on plants and animals
- But also smaller collections, e.g. wordlists with related cognacy judgements

So, hopefully, data of the kind most of you have to deal with regularly.

What is SQL?

What is SQL?

Structured **Q**uery **L**anguage

is a special-purpose programming language designed for managing data held in a relational database management system (RDBMS)

(Wikipedia)

What is a RDBMS?

A relational database is a way to store and manipulate information.
Databases are arranged as ***tables****. Each table has* ***columns*** *(also known as* ***fields****) that describe the data, and* ***rows*** *(also known as* ***records****) which contain the data.*

(software carpentry)

Database Managers

- Interacting with an RDBMS means sending SQL to a Database Manager much like interacting with the R runtime system means sending R instructions.
- The Database Manager is responsible for reading and writing the data to our disc.
- Common Database Managers include
    - Oracle (”but we are not a bank!”)
    - MySQL
    - SQLite
    - PostgreSQL

SQLite

- serverless! Just a library embedded in (**many**) programs
- The database is just a file ⇒
  - easy to backup
  - easy to share - e.g. using dropbox
- Firefox embeds SQLite, and a Firefox plugin is available as GUI for SQLite databases.
- Python supports interfacing with SQLite databases out-of-the-box, via stdlib's `sqlite3` module.
- Recommended for single user (or one user at a time).

PostgreSQL

- A client-server database $\Rightarrow$
  - requires some administion
  - supports concurrent (remote) users
- Very SQL-standard compliant (probably more so than any other RDBMS) $\Rightarrow$ easier to learn/lookup things/debug
- Scales to GB (TB even) of data $\Rightarrow$
  - so storing multimedia in the database as well is an option
  - millions of rows in a table are no problem
- Good support for many specialized datatypes, in particular GIS data

Why not Excel?

Why not Excel?

Spreadsheets - and Excel in particular

- do not scale well (once above – say – 50,000 rows, people become afraid of changing the sort order)
- do not really support typed data …
- …sometimes trying too hard to support it, like converting gene names such as `SEPT2` to dates `2006/09/02`
- are not really cross-platform (or cross-version) compatible
- do not really support complex (inter-related) data

Why SQL?

Typed data

Unlike e.g. CSV, SQL requires explicitly typed columns. Unfortunately the data type specification (or name) is not standardized across RDBMS.

- The usual suspects are typically supported: integer, float, boolean, text
- But there's a lot more:
  - Binary Large Objects (BLOBs) can be used for multimedia content.
  - JSON columns (supported in SQLite and PostgreSQL) can be used to add schemaless attribute-value-pairs to any record.
  - GIS objects like polygons (supported in SQLite and PostgreSQL)

Constraints

Constraints in SQL are rules enforced for records in a table:

NOT NULL indicates that a column cannot store NULL value

UNIQUE ensures that each row for a column (or combination of columns) must have a unique value

CHECK Ensures that the value in a column meets a specific condition (e.g. to implement enumerated types, or to make sure latitudes are numbers between -90 and 90).

DEFAULT Specifies a default value for a column

Referential Integrity and Normalization

Two constraints are used to relate records between tables in a way that ensure referential integrity:

PRIMARY KEY
: Ensures that a column (or combination columns) have a unique identity (implies NOT NULL and UNIQUE)

FOREIGN KEY
: Declare a column (or combination of columns) as ”pointer” to a primary key of another record.

The typical/simplest use case for this are reference tables, e.g.

- a LANGUAGE table with ISO 639-3 codes or Glottocodes as primary key
- a SOURCE table referenced by BibTeX citation key

ACID Transactions

While not strictly a part of SQL, both – SQLite and PostgreSQL provide transactions with guaranteeing the ACID properties:

Atomicity Transactions are **all or nothing**.

Consistency Successful transactions cannot result in an invalid database state.

Isolation Transactions are isolated from one another, i.e. to the outside world they appear as happening in a series.

Durability If a transaction is **committed**, the new state is durable, i.e. written to disc.

Other advantages of SQL

- Every software developer should know SQL, so when your data is in an RDBMS, they will know how to access it.
- SQL allows you to bundle data and schema (i.e. the data description) in a single text file.

Best practices for data modeling

- Always use auto-incrementing integer primary keys – model natural (or surrogate, or composite) candidates for primary keys via unique constraints instead. ⇒
    - Changing the unique properties later is easier.
    - The schema is more predictable/intuitive (foreign keys must be integers, too).
- Choose and adhere to naming conventions for
    - tables (**either** plural **or** singular nouns)
    - primary keys
    - foreign keys (e.g. prefix with name of referenced table)
- Use and link to standard reference catalogs
    - Glottolog (maybe ISO 639-3) for languages
    - Concepticon for concepts in wordlists
    - D-PLACE for societies
    - Any other in your field?

SQL and text data

SQL databases are not particularly good at handling non-ASCII text - but better than many other environments

- They don’t pretend text can contain markup (like Filemaker or Excel).
- They don’t support unicode in a meaningful way out of the box, but can be taught to do a little better.

Best practices for text data

- Declare the text encoding for your databases explicitly

SQLite:
```
PRAGMA encoding = "UTF-8"
```

PostgreSQL:
```
createdb -E UTF-8 ...
```

Note: The default encoding is your locale!

- Normalize your unicode data before insertion into the database, e.g. using Python's `unicodedata` module:

```
>>> unicodedata.normalize('NFC', u'\u0061\u0301')
u'\xe1'
```

- Or enable Unicode collation support via DUCET

```
# SELECT E'\u0061\u0301' = E'\u00e1';
 f

# SELECT collkey(E'\u0061\u0301', 'root', true, 4, false) =
     collkey(E'\u00e1', 'root', true, 4, false);
 t
```

SQL and version control

Given the power of SQL, it's not surprising that various schemes of keeping a history of changes in a database exist, e.g. using *triggers*.

Here's a simpler recipe, exploiting the fact that

- SQL commands are lines of text suitable for version control with tools like git
- SQL includes a DDL, i.e. a data description language describing the schema

SQL and version control

1. Dump your database to plain SQL (including the schema)

   SQLite:
   ```
   sqlite3 db.sqlite .dump > db.sql
   ```
   PostgreSQL:
   ```
   pg_dump -x db -f db.sql
   ```
2. Version control the dumped SQL file.
3. Start editing after restoring the database from the dump

   SQLite:
   ```
   sqlite3 db_restored.sqlite < db.sql
   ```
   PostgreSQL:
   ```
   createdb db_restored
   psql -d db_restored -f db.sql
   ```

Applicable if

- data isn't too big ($< 100,000$ rows, no blobs)
- no concurrent editing happens (although merging may work, resolving conclicts by hand may be no fun)

SQL and version control

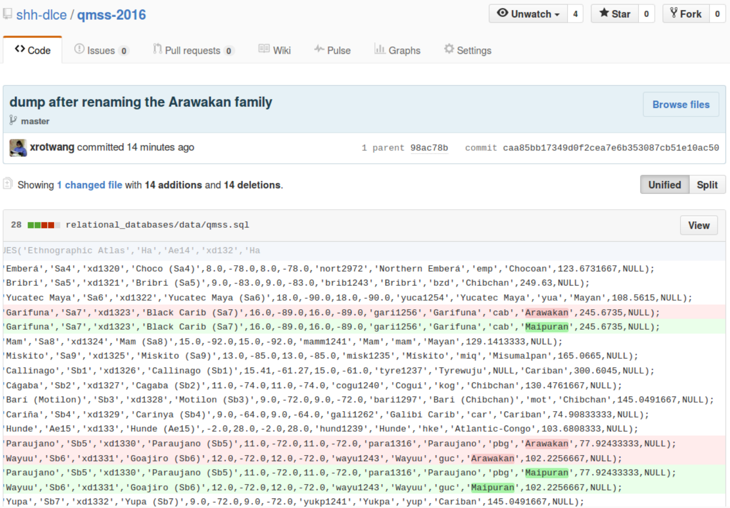

Figure 1: GitHub's display of a `diff` of a SQL dump.

Weaknesses of SQL databases

- Manual data entry is not supported too well out of the box
    - generic frontend tools are often clunky
    - custom frontends, e.g. Django apps, require programming/maintenance/administration.
- SQL dumps are not really portable across SQL implementations, thus, while a good format for versioning, these are not good for data sharing.

Conclusion I

You may go straight from collecting data in Excel to analysis in R, but …

- if you are going to collect (aggregate)
  - a lot of data
  - or over a long time
  - or with multiple collaborators
- or if analysis must be efficient/performant
- or if you want to make your data easy to repurpose/accessible from various tools/languages

…introducing SQL as data curation layer may be worth the effort!

Conclusion II

- If you find yourself experiencing the pain points of Excel already
- or if you write code to check the consistency of your data

…consider writing code instead to import your data into a SQL database with a properly constrained data model.