

# Introduction to Data Access and Storage

Thomas Rosenthal - DSI @ UofT

Module 05

# Expanding your Database:

## INSERT, UPDATE, DELETE

## Importing and Exporting Data

## Normal Forms

## Views

## Run SQLite in python and/or R

# Expanding your Database:

## INSERT, UPDATE, DELETE

## Importing and Exporting Data

## Normal Forms

## Views

## Run SQLite in python and/or R

# INSERT, UPDATE, DELETE

Prior to this, we've focused solely on retrieving values from tables:

- Tables can also be manipulated with `INSERT`, `UPDATE`, and/or `DELETE`
- *A word of warning...these commands are PERMANENT* 😱
  - Generally, follow a policy that avoids altering data
  - Make backups of tables before you run a query
  - Never hurts to test on a temporary table first!
- But they are useful, and sometimes the correct solution
- There is no `SELECT` statement for these types of queries

# INSERT, UPDATE, DELETE

## INSERT

- `INSERT` allows you to add a record
- Specify where you want to add:
  - `INSERT INTO [some_table_name]`
- ...and what you want to add:
  - `VALUES(column_one_value, column_two_value)`
- `VALUES` come in the order of the columns within the tables
- `VALUES` must respect table constraints
  - e.g. NULLs, UNIQUE, data types, etc
- `INSERT` can help create small helper tables
  - **Can we think of any scenarios?**

# INSERT, UPDATE, DELETE

## UPDATE

- UPDATE allows you to change a record
- Specify where you are making your change:
  - UPDATE [some\_table\_name]
- ...and what you want to change:
  - SET column\_one = value1, column\_one = value2
- *SPECIFY A WHERE CONDITION*
  - WHERE condition
- You can change a single column, a few columns, all the columns, etc
  - (Respecting table constraints)
- **What happens if you don't specify a WHERE condition?**

# INSERT, UPDATE, DELETE

## DELETE

- **DELETE** allows you to remove a record
- Specify where you want to delete:
  - `DELETE FROM [some_table_name]`
- **SPECIFY A WHERE CONDITION**
  - `WHERE condition`
- **What happens if you don't specify a WHERE condition?!?**
- **DELETE** doesn't *remove* a table from a database
  - Instead it removes the data from it, leaving the table structure and constraints in place

# INSERT, UPDATE, DELETE

(**INSERT, UPDATE, DELETE** live coding with a TEMP TABLE)

# Expanding your Database:

## INSERT, UPDATE, DELETE

## Importing and Exporting Data

## Normal Forms

## Views

## Run SQLite in python and/or R

# Importing and Exporting Data

Import & Export

CSV

JSON

# Importing and Exporting Data

Import & Export

CSV

JSON

# Import & Export

- RDBMs allow data to flow into and out of them
  - Some processes are easy:
    - e.g. exporting a table as a CSV file
  - ...while others are complex
    - e.g. writing a CRM to a normalized data warehouse on a nightly basis
- In DB Browser for SQLite, we can make use of the following:
  - Import and export CSV files
  - Manipulate and export JSON files
- SQLite more broadly can:
  - Produce CSVs from queries (using the command line, which we won't do)
  - Connect to other programming languages

# Importing and Exporting Data

Import & Export

CSV

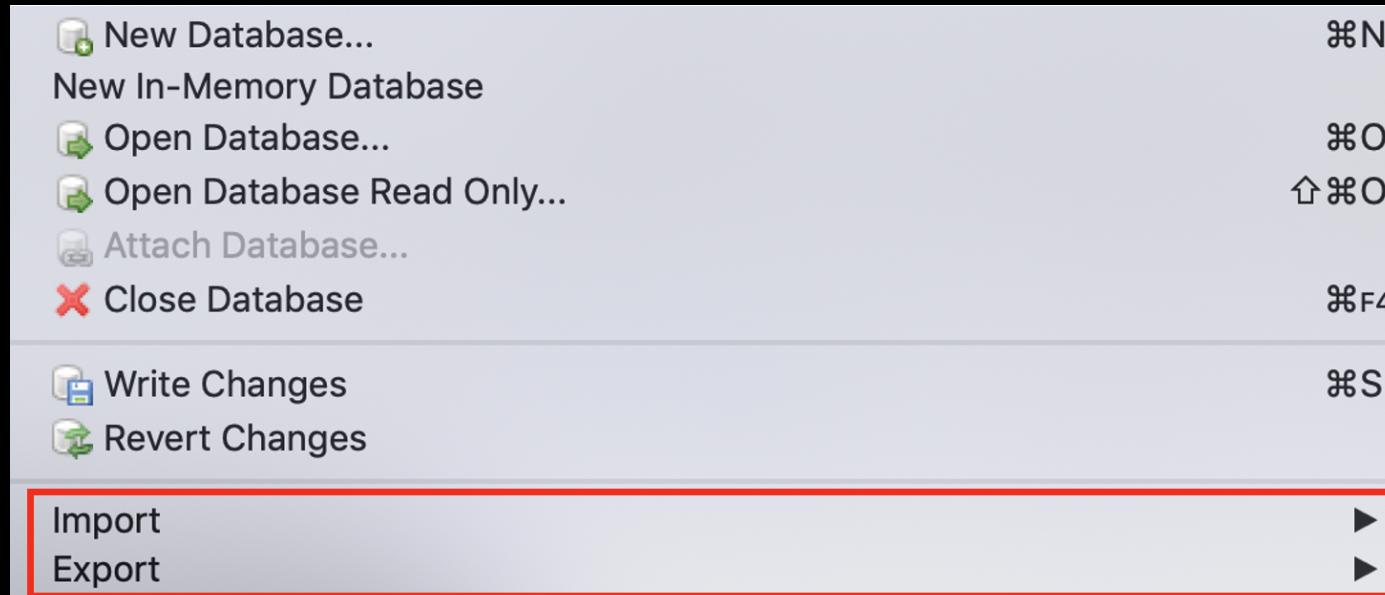
JSON

# CSV

- CSV stands for "Comma Separated Values"
- CSVs are file formats well designed to store SQL tables
  - The values of each row are separated by commas
    - Sometimes it makes more sense to use a "|" (pipe), if there is complex text data stored, which might be more sensitive to the presence of commas and/or line breaks
- They are a common file format for transporting structured data
- CSVs can be opened by:
  - Excel
  - Simple text editors (e.g. notepad++, sublime)
  - Programming languages (e.g. python, R)

# CSV

DB Browser for SQLite natively supports CSV importing and exporting for tables:



# CSV

DB Browser for SQLite allows us to extract a query result in a somewhat roundabout method:

- First, write a query

```
SELECT * FROM product p
JOIN product_category pc ON p.product_category_id = pc.product_category_id
```

- Second, right click the results, and select "Copy as SQL"
- Third, instantiate a table with CREATE

```
CREATE TABLE "example" ("product_id", "product_name", "product_size",
"product_category_id", "product_qty_type", "product_category_name") ;
```

- Fourth, paste the results from your clipboard

```
INSERT INTO "example" ("product_id", "product_name", "product_size",
"product_category_id", "product_qty_type", "product_category_name")
VALUES ('1', 'Habanero Peppers - Organic', 'medium', '1', 'lbs', 'Fresh Fruits & Vegetables');
...etc for each row
```

- Finally, export the table to CSV

# CSV

We can also extract a query result to CSV with python:

```
import pandas as pd
import sqlite3

#set your location, slash direction will change for windows and mac
DB = '/Users/thomas/Documents/GitHub/DSI_SQL/SQL/FarmersMarket.db'

#establish your connection
conn = sqlite3.connect(DB, isolation_level=None,
                      detect_types=sqlite3.PARSE_COLNAMES)

#run your query, use "\n" to allow line breaks
db_df = pd.read_sql_query("SELECT p.* ,pc.product_category_name \
                           FROM product p \
                           JOIN product_category pc \
                           ON p.product_category_id = pc.product_category_id"
                           ,conn)

#save
db_df.to_csv('database-py.csv', index=False)
```

# CSV

...or extract with R instead:

```
library(DBI)
library(RSQLite)

#set your location, slash direction will change for windows and mac
DB = '/Users/thomas/Documents/GitHub/DSI_SQL/SQL/FarmersMarket.db'

#establish your connection
conn <- dbConnect(SQLite(), DB)

#run your query
db_df <- dbGetQuery(conn, "SELECT p.* ,pc.product_category_name
                           FROM product p
                           JOIN product_category pc
                           ON p.product_category_id = pc.product_category_id")

#save
write.CSV(db_df, file = "database-R.CSV")
```

# CSV

(CSV live coding)

# Importing and Exporting Data

Import & Export

CSV

JSON

# JSON

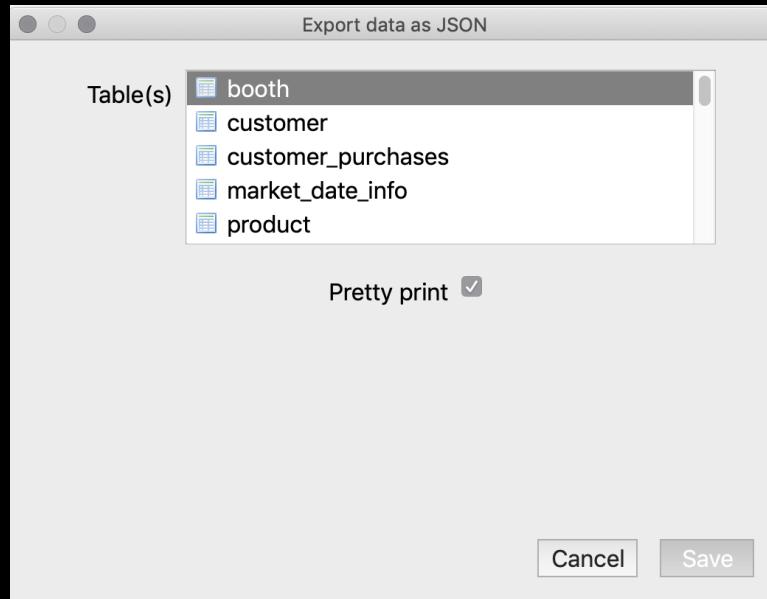
- JSON stands for "JavaScript Object Notation"
- JSONs are file formats well designed to store tables, lists, arrays, and nested objects
  - Their syntax follows specific rules:
    - Data is in name/value pairs
    - Data is separated by columns
    - Curly brackets '{ }' hold objects
    - Square brackets '[ ]' hold arrays
  - e.g. `{"first_name": "Thomas", "last_name": "Rosenthal"}`
  - or for tables:

```
[ {"first_name": "A", "last_name": "Mahfouz"}  
 {"first_name": "Thomas", "last_name": "Rosenthal"} ]
```

- JSON can be opened by:
  - Web browsers
  - Simple text editors (e.g. notepad++, sublime)
  - Programming languages (e.g. python, R)

# JSON

DB Browser for SQLite natively supports JSON exporting for tables:



# JSON

- SQLite also provides support for JSON value query and manipulation
  - This can be useful – many APIs produce JSON values, but do not parse them into tables

(json section....)

# JSON

(JSON live coding)

# Expanding your Database:

## INSERT, UPDATE, DELETE

## Importing and Exporting Data

## Normal Forms

## Views

## Run SQLite in python and/or R

# Normal Forms

Purpose

First Normal Form

Second Normal Form

Third Normal Form

# Normal Forms

Purpose

First Normal Form

Second Normal Form

Third Normal Form

# Purpose

- Databases often become more useful if they are *normalized*
  - When a table is normalized, it splits complex data stored in single columns and stores them instead in many smaller tables
  - Our farmersmarket.db is normalized
- When a collection of databases is normalized we call this a data warehouse
- There's a fine balance between the number of tables created and what is stored in any single table
  - The degree of normalization is based on criteria defined by "forms"
  - e.g. our product table could be further normalized:
    - we could place product\_qty\_type into its own table and reference it with an id

# Normal Forms

Purpose

First Normal Form

Second Normal Form

Third Normal Form

# First Normal Form

- First Normal Form (1NF) requires that each column contain one single value
- Many tables are in 1NF by default

Consider a non-normalized table:

name	OS	software	supervisor
A	win	VSCode, MSSQL, RStudio	Eric Yu
Thomas	mac	Spyder, SQLite, RStudio	Rohan Alexander

We can shift it into 1NF by unpivoting the software column:

name	OS	software	supervisor
A	win	VSCode	Eric Yu
A	win	MSSQL	Eric Yu
A	win	RStudio	Eric Yu
Thomas	mac	Spyder	Rohan Alexander
Thomas	mac	SQLite	Rohan Alexander
Thomas	mac	RStudio	Rohan Alexander

# Second Normal Form

- Second Normal Form (2NF) requires that each non-key column is dependent on the primary key
  - Therefore, no row deletions can affect the integrity of another table
- Our farmersmarket.db is 2NF!

Our table is currently 1NF, which is required for 2NF:

name	OS	software	supervisor
A	win	VSCode	Eric Yu
A	win	MSSQL	Eric Yu
A	win	RStudio	Eric Yu
Thomas	mac	Spyder	Rohan Alexander
Thomas	mac	SQLite	Rohan Alexander
Thomas	mac	RStudio	Rohan Alexander

# Second Normal Form

2NF requires that we prevent supervisors from being deleted, should A or myself leave the school.

Supervisors now becomes its own table:

<b>id</b>	<b>name</b>
1	Eric Yu
2	Rohan Alexander

# Second Normal Form

We introduce a PK id and supervisor\_id becomes our FK, replacing supervisor:

<u>id</u>	<u>name</u>	<u>OS</u>	<u>supervisor_id</u>
1	A	win	1
2	Thomas	mac	2

...and student\_software now references that PK:

<u>id</u>	<u>software</u>
1	VSCode
1	MSSQL
1	RStudio
2	Spyder
2	SQLite
2	RStudio

# Third Normal Form

- Third Normal Form (3NF) requires that we replace any non-key transitive functional dependency
  - This means if any non-key column's value changed, and therefore another non-key value would be invalidated, we must replace this dependency with a table relationship instead

Our table is currently 2NF, which is required for 3NF:

<b>id</b>	<b>name</b>	<b>OS</b>	<b>supervisor_id</b>
1	A	win	1
2	Thomas	mac	2

# Third Normal Form

Because MSSQL is only available on Windows, any change in OS will change whether MSSQL can be installed

OS must become its own table:

OS_id	OS	win_only
1	win	TRUE
2	mac	FALSE

...and we must create a software table referencing it

software_id	software	win_only
1	MSSQL	TRUE
2	RStudio	FALSE
3	VSCode	FALSE
4	SQLite	FALSE
5	Spyder	FALSE

# Third Normal Form

At last, we have 3NF, for both our student\_software table:

<u>id</u>	<u>OS_id</u>	<u>software_id</u>
1	1	1
1	1	2
1	1	3
2	2	2
2	2	4
2	2	5

...and our original table:

<u>id</u>	<u>name</u>	<u>OS_id</u>	<u>supervisor_id</u>
1	A	1	1
2	Thomas	2	2

# Normal Forms

(Normal Forms live coding, putting the previous example into action!)

# Expanding your Database:

## INSERT, UPDATE, DELETE

## Importing and Exporting Data

## Normal Forms

## Views

## Run SQLite in python and/or R

# Views

- Views instantiate a query result permanently
- They are particularly useful in highly normalized databases, where reproducing a query is tiresome or prone to query errors
- In databases that have live data flowing in:
  - Tables that are created from queries need to be continuously updated whenever there is new data
    - This requires either downtime where the table is empty
    - Or the chance of a "dirty read" (where a table is read before the data is fully updated)
  - Views, on the other hand, will always show the most up-to-date values!
- Views are created just like tables:

```
CREATE VIEW history AS  
SELECT ...
```

# Views

(Views live coding)

# Expanding your Database:

## INSERT, UPDATE, DELETE

## Importing and Exporting Data

## Normal Forms

## Views

## Run SQLite in python and/or R

# Run SQLite in python and/or R

- This module has already shown how to connect a SQLite database to python and/or R and interact with tables
- **...but did you know you can also run SQLite queries on python and/or R dataframe objects?**
- Both languages offer good support
  - Often there is corresponding python and/or R syntax to achieve similar results
    - As always with data, there is no prescribed way of doing something!

# Run SQLite in python and/or R

## Python Example

```
import pandas as pd
import pandasql as sql #this allows us to run SQLite queries!

p = "https://raw.githubusercontent.com/allisonhorst/palmerpenguins/master/inst/extdata/penguins.csv"
penguins = pd.read_csv(p) #create a dataframe
yrly_penguins = sqlssql(''SELECT DISTINCT year, COUNT(*) AS count,
                           SUM(COUNT(*)) OVER (ORDER BY year) AS running_total
                           FROM penguins
                           GROUP BY year'''') #run a SQLite query with sqldf()
```

---

year	count	running_total
2007	110	110
2008	114	224
2009	120	344

---

# Run SQLite in python and/or R

## R Example

```
library(palmerpenguins) #load some data
library(sqldf) #this allows us to run SQLite queries!

penguins <- penguins #create a dataframe
avg_mass <- sqldf('SELECT DISTINCT species, sex, AVG(body_mass_g) as avg_mass
  FROM penguins
  WHERE sex <> "NA"
  GROUP BY species, sex') #run a SQLite query with sqldf()
```

---

species	sex	avg_mass
Adelie	female	3368.836
Adelie	male	4043.493
Chinstrap	female	3527.206
Chinstrap	male	3938.971
Gentoo	female	4679.741
Gentoo	male	5484.836

