Homework - Data Management and SQLite

Lachlan Deer   Adrian Etter   Julian Langer
Max Winkler

Introduction

This set of guided exercises provides you with some additional problems you can work through to gain more experience using SQLite and the principles of Tidy Data.

Exercises

Exercise 1 - Importing data to a new database

The data are stored in `examples/householdWealth/code-and-data/data` as a collection of csv files. The first task is to import these into a SQL database. To get us started *slowly* we will only import three tables: asset, household and portfolio - the remainder will be left for later.

First we want to create empty tables to store the data. We will initially create three tables, one for each data set, called "asset", "household", and "portfolio". First inspect each data set using the bash command `head` so you know what variable names are included in the data.

Begin a SQL script `01-createAndImport.sql`. The first part of your script should create table templates by adapting the following code:

```
CREATE TABLE tableName
(
      variable1         TYPE
    , variable2         TYPE
    <...>
    , PRIMARY KEY (variableX)
)
;
```

Twist: In the asset data set there is no obvious Primary Key since we are observing difference assets over time. Replace `PRIMARY KEY (variableX)` with

`UNIQUE(AssetID, Time)` so that each observation has a unique identifier.

Once you have created the tables, its time to import the data. In this excercise we are going to take a slightly different approach to importing data than we did in class. This is because sometimes it works better, and gives you less cryptic error messages. We are going to import each csv into a temporary table we will call `tempItem` and then take that table and extract the data we need into our table structures we just created before deleting the `tempItem`. Place this code *after* the table creation lines of `01-createAndImport.sql` Below we provide an example of how to do this for the asset data:

```
.import ./data/asset.csv tempItem

INSERT INTO asset
(
      AssetID
    , Time
    , LogReturn
)
SELECT
      AssetID
    , Time
    , LogReturn
FROM
    tempItem
;

DROP TABLE
    tempItem
;
```

Repeat this code for each table, and place all the import commands into a file called ‘importData.sql.’

You have now successfully imported some of the data.

Exercise 2: Simple Queries

We are now ready to use the data that we have to conduct some simple queries using the (SELECT, FROM, WHERE) logic. Like we did in our session, we structure the queries as:

```
SELECT
<...>
FROM
<...>
```

```
WHERE
<...>
```

1. Write a simple query that returns each household's ID alongside it's wealth as a csv file. Call the query `02-01-displayWealth.sql`. If your solution is correct, you output will look similar to (difference is the comma separated formatting):

   ```
   HouseholdID  Wealth
   -----------  ----------
   1            196087.3
   2            316478.7
   3            294750.0
   ```

2. Write a query that returns the household ID and wealth for the household who is identified by the ID number '2.' Call the query `02-02-displayWealthForHouseholdID.sql`. If your solution is correct, you will see the following output as a result:

   ```
   household_id  wealth
   ------------  ----------
   2             316478.7
   ```

3. Write a query that returns the household ID and wealth for all households where a female is the household head. Call the query `02-03-displayWealthForCondition.sql`. If your query is correct, you will see the following output as a result:

   ```
   household_id  wealth
   ------------  ----------
   1             196087.3
   3             294750.0
   ```

This completes our examples on simple queries.

Exercise 3 - Inserting new data

Suppose you now are provided information via email about a new individual who was omitted from the data set. We want to include his information into your data. We will assign the individual an ID of 4, he is male, and has a wealth of 261534.3.

Go back the the slides where we demonstrated how to add a row of data to a table. Use that format to input this new individuals data into the `households` table. Call the script in which you perform the operation `03-insertNewData.sql`. If you did this correctly, the households table should have the following information inside:

```
HouseholdID  Male        Wealth
-----------  ----------  ----------
1            0           196087.3
2            1           316478.7
3            0           294750.0
4            1           261534.3
```

Excercise 4 - Importing Data Redux

We want to import an additional data set into the database, socioDemog.csv. Repeat the type of commands from Exercise 1 to import this table into the database sing a script called `04-importSocioDemog.sql`. If you did this successfully, you should be able to enter the comand `.tables` into sqlite and see the following output

```
.tables

asset household portfolio socioDemog
```

Exercise 5 - One to One Merges

We know want to merge two different data tables together using the INNER JOIN command. We want to construct a table (and export it) that links together the information in the households table to the socio demographic information we just imported into the database. The standard structure for such a query would be:

```
SELECT
    <...>
FROM tbl1, tbl2

INNER JOIN tbl1
    ON tbl1.variable = tbl2.variable
;
```

Construct a query called `05-oneToOneMerge.sql` that links together the data in the household table and the socio-demographic information using the structure above. If your query is successful, you should get the following output:

```
HouseholdID  Male        Kids        Married     Wealth
-----------  ----------  ----------  ----------  ----------
1            0           1           0           196087.3
2            1           1           1           316478.7
3            0           0           0           294750.0
4            1           0           1           261534.3
```

Exercise 6 - Many to One Merges

Now we want to conduct a query that allows us to link the household information in the households table to each household's investment portfolio. Unlike some statistical softwares, you can conduct many to One merges using the same syntax as the one to one merge.

Link together all the information from the households table with the asset ID, asset Name and Share of each asset for the first two households. Call the query `06-manyToOneMerge.sql`. If done correctly, the resulting merged data should like:

```
HouseholdID  AssetID       Male        Wealth      Name        Share
-----------  ------------  ----------  ----------  ----------  ----------
1            nl0000301109  0           196087.3    ABN Amro    1.0
2            gb00b03mlx29  1           316478.7    Royal Dutc  0.6
2            nl0000289783  1           316478.7    Robeco      0.4
```

Exercise 7 - Many to Many Merges

Finally we want to construct a merged data set that has for each household ID, information about the investment portfolio and the log returns of each asset for each period of time.

Using the (hopefully) familar syntax, create a query called `07-manyToManyMerge.sql` that links all this information together. If done successfully, the first few lines of the output should look like:

```
HouseholdID  AssetID       Time        Name        Share       LogReturn
-----------  ------------  ----------  ----------  ----------  -----------
1            nl0000301109  9           ABN Amro    1.0         -0.10881527
1            nl0000301109  10          ABN Amro    1.0         0.00968313
1            nl0000301109  11          ABN Amro    1.0         -0.01289824
1            nl0000301109  12          ABN Amro    1.0         0.02563224
1            nl0000301109  13          ABN Amro    1.0         0.08774676
1            nl0000301109  14          ABN Amro    1.0         0.09400246
```

OK, we have the ideas behind merges down so it is time to change gears a little!

Exercise 8 - Tidy Tables

Now that we have imported the socioDemog table into our database the `households` table we initially imported is a bit of a misnomer and needs to be worked on:

- Male is a socio-demographic variable and needs to be moved into the `socioDemog` table

- the table `households` can then be renamed to something more descriptive `wealth`

To do both of these tasks we will proceed step by step. First create a script called '08-moveMale.sql' which will perform these tasks.

To add data to a table we need to use the `ALTER TABLE` command. In order to copy the 'male' variable across to socioDemog, copy the following code into this file:

```
ALTER TABLE socioDemog
    ADD COLUMN
        Male INTEGER NOT NULL DEFAULT '999'
;

UPDATE socioDemog
    SET Male =
    (
        SELECT
            households.Male
         FROM
            households
         WHERE
            socioDemog.HouseholdID = households.HouseholdID
    )
;
```

Convince yourself that you understand what this code does - it's not too mysterious.

Next, we want to remove the 'male' variable from the households table and then rename the table. Unfortunately SQLite does not have a drop variable command (other SQL variants do). Instead we can just create a new table called `wealth` that contains the relevant varaibles from the `household` table, and then use the `DROP TABLE` command to drop the households tables from the database.

The following code does this for you, paste it into the '08-moveMale.sql' script and ensure you understand what it is doing.

```
CREATE TABLE wealth
(
    HouseholdID       INTEGER NOT NULL
    , Wealth            FLOAT NOT NULL
    , PRIMARY KEY (HouseholdID)
)
;

INSERT INTO wealth
(
      HouseholdID
    , Wealth
)
SELECT
      HouseholdID
    , Wealth
FROM households
;

DROP TABLE
    households;
```

We leave it to you to verify this does what we wanted it to do: that the table `household` no longer exists, and that the table `wealth` has the information on wealth for each household_id.

Exercise 9 - More Table Tidying

We also need to work a little on the `asset` table so that it obeys the normal forms of data outlined in the lecture slides. We will proceed as follows:

- Rename the asset table to a more meaningful table name - `monthlyReturn`
- Create a new `asset` table that links an AssetID to the name of the asset, and whether the asset is a 'stock' or a 'fund'
- Add information on whether each asset is a stock or a fund (the current data don't make it completely clear)
- Remove the Asset's Name from the portfolio table

Convince yourself that doing this means our new table structure will obey the normal forms outlined in class.

Now let's code it up. Create a script called `09-renameAssets.sql` that will perform all of these tasks. First we want to rename the original `asset` table to monthlyReturn:

```
ALTER TABLE asset
    RENAME TO
        monthlyReturns
;
DROP TABLE asset;
```

Next we want to create a new `asset` table that relates the AssetID to the name of the asset and the type of the asset. Note that we only want to collect the **unique** Asset IDs and Names, and not have them replicated many times:

```
CREATE TABLE asset
(
      AssetID       CHARACTER(12)
```

```
    , Name          TEXT NOT NULL
    , Type          TEXT NOT NULL DEFAULT "unknown"
    , PRIMARY KEY (AssetID)
)
;

INSERT INTO asset
(
      AssetID
    , Name
)
SELECT DISTINCT
    AssetID, Name
FROM
    portfolio
;
```

Now we will add the type of each asset to the the `asset` table. Suppose we know that the assets 'ABN AMRO' and 'Royal Dutch Shell' are `stocks`, while the other assets are `funds.` We can add this information to the `asset` table using an (UPDATE,SET,WHERE,IN) command as below:

```
UPDATE asset
    SET
        Type='stock'
    WHERE
        AssetID
    IN
        ( 'nl0000301109' , 'gb00b03mlx29')
;

UPDATE asset
    SET
        Type='fund'
    WHERE
        AssetID
    NOT IN
        ( 'nl0000301109' , 'gb00b03mlx29')
;
```

Finally we want to drop the variable 'Name' from the `portfolio` table because it is just additional information and violates the 2nd normal form of data. Using the same ideas as expressed in Exercise 8, we leave it to you to remove this information from the `portfolio` table.

Exercise 10 - More Joins, Sub-Queries and Using Functions

Now we have our data in the right format let's work with it a little more to build familarity with more of SQL's commands.

The first thing we will do is revist the (SELECT, FROM, INNER JOIN, WHERE) queries. Lets create a query called `10-summStockOwners.sql` that returns a infomation only on stock owners. Namely, return a csv file that contains the following variables: HouseholdID, AssetID, Share, Type, Name.

If your query is correct, you will get the following results:

```
HouseholdID  AssetID       Share       Type        Name
-----------  ------------  ----------  ----------  ----------
1            nl0000301109  1.0         stock       ABN Amro
2            gb00b03mlx29  0.6         stock       Royal Dutc
3            gb00b03mlx29  0.15        stock       Royal Dutc
```

Now suppose we want to add to this query information on a household's wealth. There are two ways of doing this, add an extra INNER JOIN command (you can have as many successive ones as you like), or create a subquery to link the information in the `portfolio` and `wealth` tables into a temporary table `temp` and then link this table to information in the asset table before selecting on asset type.

We perform the subquery, not because its the most efficient, but because it shows a simple example of how to use them. Create a query called `10-summStockOwnersAddWealth.sql` and copy the code from below:

```sql
SELECT
      temp.HouseholdID     AS HouseholdID
    , temp.AssetID         AS AssetID
    , temp.Share           AS Share
    , asset.Type           AS Type
    , asset.Name           AS Name
    , temp.Wealth          AS Wealth

FROM asset, temp

INNER JOIN
(
    SELECT
          portfolio.HouseholdID     AS HouseholdID
        , portfolio.AssetID         AS AssetID
        , portfolio.Share           AS Share
        , wealth.Wealth             AS Wealth

    FROM portfolio
    INNER JOIN wealth
        ON portfolio.HouseholdID = wealth.HouseholdID
) temp
    ON asset.AssetID = temp.AssetID

WHERE
    asset.Type = 'stock'
;
```

Note that inside the parentheses is our subquery: we link up information in the `portfolio` and `wealth` table, and store the result for the time being in the table called `temp`. Next we link information from the `temp` table to the `asset` table, before filtering the data based on asset type.

Now that we have a some familiarity with subqueries we are going to use them in slightly more complicated examples. The first task we will complete is to select out ABN Amro returns for a selected time horizon. Place the query in `10-selectABNReturns.sql` To do this we are going to introduce the `LIKE` command. We will proceed as follows: use a subquery that utilises the `LIKE` command to select out the necessary asset ID and asset Name from the `asset table`. We will store this again in a temporary table `temp`. And then join information from `temp` to the monthlyReturn information before selecting out returns for all periods within a time range.

Here's the code that does just that:

```
SELECT
      temp.AssetName               AS AssetName
    , monthlyReturns.Time          AS Time
    , monthlyReturns.LogReturn     AS LogReturn

FROM monthlyReturns, temp

INNER JOIN
(
    SELECT
          asset.AssetID   AS AssetID
        , asset.Name      AS AssetName

    FROM asset

    WHERE
        asset.Name LIKE 'ABN%'
) temp
    ON monthlyReturns.AssetID = temp.AssetID

WHERE monthlyReturns.Time <= 12
```

```
;
```

And here is the output you expect to see:

```
AssetName   Time        LogReturn
----------  ----------  -----------
ABN Amro    9           -0.10881527
ABN Amro    10          0.00968313
ABN Amro    11          -0.01289824
ABN Amro    12          0.02563224
```

A final excercise that we will leave to you to code is to use a subquery together with some Aggregate Functions. We want to find the mean LogReturn for each asset and display that alongside the number of time periods that each asset was observed in the data. You will use a subquery to calculate these quantities before linking them to the asset Name and displaying the result in a simple table.

Call the script ‘computeAvgReturn.sql.’ It should have the following structure:

```
SELECT
    <...>

FROM tbl1, temp

INNER JOIN
(
    SELECT
          tbl2.varName1              AS newName1
        , COUNT(tbl2.varName2)       AS newName2
        , AVG(tbl2.varName3)         AS newName3

        FROM tbl3

        GROUP BY
            tbl3.varNameX
) temp
    ON tbl1.varNameY = temp.varNameZ

GROUP BY
    tbl1.varName
```

It is left to you to replace the ‘placeholder’ names with the right ones. If you do this correctly, here is the result you will get:

```
Name               NumberObs   AvgLogReturn
-----------------  ----------  -------------------
Royal Dutch Shell  235         0.00676829880851064
AAB Eastern Europ  48          -0.0007689702083333
Robeco             235         0.00365037834042553
```

```
Postbank BioTech   64          -0.0009290095312500
ABN Amro           212         0.0144418989150943
```

That's all folks!

Happy SQL adventures!