US Houshold Income Data Cleaning

Open up the .csv files with Microsoft Excel and take a quick look
at what sort of data are in the files. Alex mentions a data dictionary
when talking about the data, but that dictionary isn't available for this lesson.

Alex points out there are over 32,000 rows in one of the files associated with this project.

Create a new schema and import the first dataset using the Table Data Import Wizard
after right-clicking Tables under the new schema in the navigator.

Alex points out that MySQL is usually good at correctly identifying
the field types for the columns of data coming in. Alex also points out that some of the rows
in the US household income file weren't transferred over to the MySQL Workbench. He says that
the viewer can go back to the file and look for rows with null values; however, considering how
the vast majority of the rows were imported into the database, Alex decides to proceed forward.
I suppose the decision to go back to investigate dropped rows will depend on
circumstances, but because any missing rows can actually be important to the final analysis,
I'd typically be inclined to go back to try to see what got dropped.
If there's really no way to figure out what the missing
values could be, then it might be better to let those rows remain missing since a bad guess of a missing value
could potentially be also bad for the final analysis. I could then at least say that I had looked into the matter,
and the presence of null data could then maybe be attributed to a fault in data collection techniques which I
would have limited control over if any.

At any rate, I'll continue to follow along with Alex. He refreshes all in the navigation window and looks at
the full table for US household income. Everything looks good acocrding to him, so he moves on to import the next
set of data.

Note: It may seem excessive to document my thoughts during this video; however, actively thinking through
what's going in the video can potentially pay off in massive dividents in the future because in the end
I'll have to think as an analyst.

Alex points out that it looks like fewer rows had been dropped during the import of the US household income statistics
file than during the import of the US household income file. Again, Alex decides to move forward, and once again
I'll follow along. He refreshes all once again in the navigation pane and looks at the full table for the
US household income statistics.

He then copies the queries that look at the full tables into the main SQL file so that
things are in one place.

```
SELECT *
FROM us_project.us_household_income
;

SELECT *
FROM us_project.us_household_income_statistics
;

# Alex notices that one of the column names of the table for US household income statistics
# needs to be updated.

# Out of curiosity, I reviewed the video on data cleaning for the world life expectancy project,
# and I noticed that Alex creates a backup table before deleting any content from the main table.
# I suppose the same approach applies here, too - make changes as needed to the main table, but then
# create a backup table before removing anything.

ALTER TABLE us_project.us_household_income_statistics
RENAME COLUMN `ï»¿id` TO `id`
;

# Review the US household income statistics table
# to confirm that the change had gone through as intended.

SELECT *
FROM us_project.us_household_income_statistics
;

# Determine the count of IDs in each table.
SELECT COUNT(id)
FROM us_project.us_household_income
;

SELECT COUNT(id)
FROM us_project.us_household_income_statistics
;

# Alex points out that it is very common for MySQL
# to drop rows when importing data. He also points out that
# this is a very real dataset that he downloaded from a
# government website, so there will potentially be issues with it.
# He opts not to fix the data before importing it; instead,
# he'll just go with the raw data. I'll follow along with him
# for the time being but then use my best judgement in the future
# when deciding to clean up the data before importing them into MySQL.

# Alex reviews the full tables once more.
SELECT *
FROM us_project.us_household_income
```

```
;

SELECT *
FROM us_project.us_household_income_statistics
;

# At first glance, Alex notices that the full state names can show up
# with a lower-case first letter; he also notices that some rows will have zeros
# for numerical values (maybe because of a lack of proper reporting).

# Look at the US household income table to see instances of IDs that appear
# more than once.
SELECT id, COUNT(id)
FROM us_project.us_household_income
GROUP BY id
HAVING 1 < COUNT(id)
;

# Look at the full table for US household income to verify that the individual rows
# have their own row IDs. This will help with cleaning data.
SELECT *
FROM us_project.us_household_income
;

# Isolate the row IDs for dupilcate rows by creating row numbers over the data
# partitioned by the column titled id, and then isolating the rows by partition row numbers greater than
one.
SELECT *
FROM (
SELECT row_id,
            id,
   ROW_NUMBER() OVER(PARTITION BY id ORDER BY id) AS row_num
FROM us_project.us_household_income
) AS duplicates
WHERE 1 < row_num
;

# Repeat the above query but focus on just the row IDs (not to be confused
# with partition rown numbers - yes, this can be a bit confusing).
SELECT row_id
FROM (
SELECT row_id,
            id,
   ROW_NUMBER() OVER(PARTITION BY id ORDER BY id) AS row_num
FROM us_project.us_household_income
) AS duplicates
WHERE 1 < row_num
;
```

```
# Alex does not appear to create a backup table of the original data prior to delting rows,
# so I'll take the initiative to make a backup table myself using the table data import wizard.
# My backup table is titled us_household_income_backup.

# I'll run a "SELECT version" of Alex's DELETE query in order to confirm for myself that
# I'll be deleting the right rows.
SELECT *
FROM us_household_income
WHERE row_id IN (
	SELECT row_id
	FROM (
	SELECT row_id,
		id,
		ROW_NUMBER() OVER(PARTITION BY id ORDER BY id) AS row_num
	FROM us_project.us_household_income
	) AS duplicates
	WHERE 1 < row_num
)
;

# The above query looks good, so I'll go forward the the DELETE query.
DELETE
FROM us_household_income
WHERE row_id IN (
	SELECT row_id
	FROM (
	SELECT row_id,
		id,
		ROW_NUMBER() OVER(PARTITION BY id ORDER BY id) AS row_num
	FROM us_project.us_household_income
	) AS duplicates
	WHERE 1 < row_num
)
;

# Check that there are no more duplicate rows.
SELECT *
FROM (
SELECT row_id,
	id,
   ROW_NUMBER() OVER(PARTITION BY id ORDER BY id) AS row_num
FROM us_project.us_household_income
) AS duplicates
WHERE 1 < row_num
;

# Next is to delete duplicates from the US household income statistics table.
```

```
# Start by looking at the US household income statistics table to see instances of IDs that appear
# more than once.
SELECT id, COUNT(id)
FROM us_project.us_household_income_statistics
GROUP BY id
HAVING 1 < COUNT(id)
;

# No duplicate rows show up in the result of the above query, so there's
# no need to take further steps to delete anything from the
# US household income statistics table.

# Look at the whole US household income table again
# to begin addressing state names that begin with lower-case letters.
SELECT *
FROM us_project.us_household_income
;

# Focus on the state names of the US household income table.
SELECT State_Name, COUNT(State_Name)
FROM us_project.us_household_income
GROUP BY State_Name
;

# There's an instance of 'georia' which looks like a misspelling of
# 'Georgia.' However, it looks like MySQL treats 'alabama' the same as
# 'Alabama' (i.e., MySQL doesn't appear to be case-sensitive), and this concerns
# Alex because he wants to know if there any spelling mistakes.

# Try looking at distinct state names in ascending order.
# Reminder: MySQL appears to be 1-indexed.
SELECT DISTINCT State_Name
FROM us_project.us_household_income
ORDER BY 1
;

# Based off of the result of the above query, it looks like 'alabama'
# doesn't show up as something different from 'Alabama.' When performing
# aggregate functions, both upper-case and lower-case spellings will likely
# be aggregated together; however, Alex says that he perfers to have everything standardized
# which is understandable.

# Alex tries to change 'georia' to 'Georgia.'
UPDATE us_project.us_household_income
SET State_Name = 'Georgia'
WHERE State_Name = 'georia'
;
```

```
# Look at the state names again.
SELECT DISTINCT State_Name
FROM us_project.us_household_income
ORDER BY 1
;

# Based off of the above query, it looks like the update occurred
# as expected.

# Alex then tries to change 'alabama' to 'Alabama.'
UPDATE us_project.us_household_income
SET State_Name = 'Alabama'
WHERE State_Name = 'alabama'
;

# Look at the whole US household income table again
# to confirm the above change because the data value 'alabama'
# was first found when looking at the whole table.
SELECT *
FROM us_project.us_household_income
;

# And based off of the result of the above query, it looks
# like the change did go through as intended.

# Alex then looks through the result of the above query
# for any other issues worth addressing.

# Alex decides to investigate state abbreviations.
SELECT DISTINCT State_ab
FROM us_project.us_household_income
ORDER BY 1
;

# And everything looks good to Alex based off of the result
# of the above query.

# Look at the whole US household income table once more.
SELECT *
FROM us_project.us_household_income
;

# Alex decides to investigate rows where values in the Place column are blank
# Note: Alex says that he's looking for null values, but based off of the query
# that he uses it looks like he's looking for blank values which are different
# Here's a source of some information on the
# difference: https://stackoverflow.com/questions/46066249/what-are-the-differences-between-null-zero-and-blank-in-sql
```

```
SELECT *
FROM us_project.us_household_income
WHERE Place = ''
ORDER BY 1
;

# Only one row appears in the result of the above query

# Alex then invesgiates rows with the same county as this result
SELECT *
FROM us_project.us_household_income
WHERE County = 'Autauga County'
ORDER BY 1
;

# Based off of the result of the above query, the vast majority of rows with
# Autauga County as the county also have Autaugavill as the associated place. (Only one row
# has a different place - row ID 147 with Pine Level for place.) Out of curiosity, I did a Google search
# on Pine Level, Alabama. Here's a source for some information on the
# place - https://en.wikipedia.org/wiki/Pine_Level,_Autauga_County,_Alabama

# Alex decides to fill in the cell with the null value in the result of
# the above query with a new value based off of the
# row's county value and city value.
UPDATE us_household_income
SET Place = 'Autaugaville'
WHERE County = 'Autauga County'
AND City = 'Vinemont'
;

# Alex then runs the query below to confirm the change.
SELECT *
FROM us_project.us_household_income
WHERE County = 'Autauga County'
ORDER BY 1
;

# And it looks things are set as intended.

# Alex decides to look at the Type column next
# Reminder: Include the GROUP BY statement when using the COUNT() function.
SELECT `Type`, COUNT(`Type`)
FROM us_project.us_household_income
GROUP BY `Type`
;

# Based off of the result of the above query, there's a 'Borough' type and a
# 'Boroughs' type. Also, there's a 'CDP' type and a 'CPD' type. Alex did mention earlier that there's a
```

```
# data dictionary associated with this data, but maybe there's nothing in it that explains what
# the types actually mean. Otherwise, I'd imagine he'd mention it to the viewer for instructional
purposes.
# It's not clear if 'CPD' and 'CDP' really are two different things or if there's a typo involved. However,
# 'Boroughs' does seem like a typo, so Alex changes any instances of 'Boroughs' to just 'Borough.'
# Note: The word Type actually is considered a keyword in MySQL, so I'm deliberately using the grave
accent (`)
# around that word to make it clear that I want the column title and not the keyword.
UPDATE us_household_income
SET `Type` = 'Borough'
WHERE `Type` = 'Boroughs'
;

# Review the counts of the different types.
SELECT `Type`, COUNT(`Type`)
FROM us_project.us_household_income
GROUP BY `Type`
;

# Based off of the result of the above query, everything looks good.

# Review the whole table again to see if there's anything else to clean.
# Alex comments how the data cleaning process can be a bit arduous or long,
# but he also says that that's okay.
SELECT *
FROM us_project.us_household_income
;

# Alex would like to look at ALand and AWater next because there were some zeros there.
SELECT ALand, AWater
FROM us_project.us_household_income
WHEre AWater = 0 OR AWater = '' OR AWater IS NULL
;

# And it looks like there are a lot of zeros under AWater.

# Alex then decides to look at distinct numbers in the AWater column to see if there are any blanks or
nulls
SELECT DISTINCT AWater
FROM us_project.us_household_income
WHERE AWater = 0 OR AWater = '' OR AWater IS NULL
;

#And it looks like there are only zeros in the result; no blanks or nulls present.

# Alex then invesitages instances of zeros, blanks, or nulls in the ALand column as well as
# zeros, blanks, or nulls in the AWater column
SELECT ALand, AWater
```

```
FROM us_project.us_household_income
WHERE (AWater = 0 OR AWater = '' OR AWater IS NULL)
	AND (ALand = 0 OR ALand = '' OR ALand IS NULL)
;

# And nothing shows up.

# Alex then invesitages instances of zeros, blanks, or nulls in just the ALand column.
SELECT ALand, AWater
FROM us_project.us_household_income
WHERE (ALand = 0 OR ALand = '' OR ALand IS NULL)
;

# And it looks like some rows do show up, which indicates that some areas are just all water

# Alex comments that all the cleaning that needs to be done has been done. There wasn't a whole lot to do, but
# he does mention that circumstances can require a lot of data cleaning. Then he goes on to review
# all the different cleaning steps that had been taken over the course of this video. He also says that he acquired
# the data from the government and praises them for collecting and organizing the data in such a way
# that not a lot of data cleaning needed to be done.
```