Website (https://www.coleridgeinitiative.org)

Ghani, Rayid, Frauke Kreuter, Julia Lane, Adrianne Bradford, Alex Engler, Nicolas Guetta Jeanrenaud, Graham Henke, Daniela Hochfellner, Clayton Hunter, Brian Kim, Avishek Kumar, Jonathan Morgan, Ursula Kaczmarek, Benjamin Feder.

source to be updated when notebook added to GitHub

Dataset Exploration

Table of Contents

JupyterLab contains a dynamic Table of Contents that can be accessed by clicking the last of the six icons on the left-hand sidebar.

Introduction

In an ideal world, we have all of the data we want with all of the desirable properties (no missing values, no errors, standard formats, and so on). However, that is hardly ever true, and we have to work with using our datasets to answer questions of interest as intelligently as possible.

In this notebook, we will explore some of the datasets we have on the ADRF to answer some questions of interest.

Learning Objectives

This notebook will give you the opportunity to spend some hands-on time with the data. We will base our discussions around the following questions:

What are WIC households' total food expenditures in 2017? What is the share of WIC purchases for these households?

These questions provide the framework for *Sample Project 1*, which you can access in the shared folder on the ADRF.

Within the scope of the questions, you will have an opportunity to explore different datasets in the ADRF, and this notebook will show you some ways you can analyze your data. This involves looking at basic metrics in the larger dataset, taking a subset of the data, creating derived variables, making sense of other variables, and so on.

This will be done using both SQL and `pandas` (a Python package). The `pyathenajdbc` Python package provides a connection to Athena to pull data into Python.

This notebook will provide an introduction and examples for:

- How to create new tables from the larger tables in database (sometimes called the "analytical frame")
- How to explore different variables of interest
- How to explore aggregate metrics
- How to handle type conversions
- How to join newly created tables

Methods

We will be using the `pyathenajdbc` Python package to access tables in our data store (Athena).

To read the results of our queries, we will be using the `pandas` Python package, which has the ability to read tabular data from SQL queries into a pandas DataFrame object. Within `pandas`, we will use various commands:

- Subsetting data
- `read_sql`
- `head`

Within SQL, we will use various queries to:

- Select data subsets
- Sum over groups
- Create new tables
- Count distinct values of desired variables
- Group data by chosen variables
- Select a sub-sample of the data

Python Setup

In Python, we `import` packages. The `import` command allows us to use libraries created by others in our own work by "importing" them. You can think of importing a library as opening up a toolbox and pulling out a specific tool. Some Python packages include:

- `numpy` is short for "numerical Python". `numpy` is a lynchpin in Python's scientific computing stack. Its strengths include a powerful *N*-dimensional array object and a large suite of functions for doing numerical computing.
- `pandas` is a library in Python for data analysis that uses the DataFrame object (modeled after R DataFrames, for those familiar with that language) which is similiar to a spreedsheet but allows you to do your analysis programaticaly rather than the point-and-click of Excel. It is a lynchpin of the PyData stack and is built on top of `numpy`.
- `pyathenajdbc` is a Python library for interfacing with Athena.

In [ ]:
```
# pandas-related imports
import pandas as pd

# Athena interaction imports
from pyathenajdbc import connect
```

When in doubt, use shift + tab to read the documentation of a method.

The `help()` function provides information on what you can do with a Python function.

In [ ]:
```
## for example
help(pd.read_sql)
```

Access the Data

We can execute SQL queries using Python to get the best of both worlds. For example, Python - and pandas in particular - make it much easier to calculate descriptive statistics of the data. Additionally, as we will see in the Data Visualization exercises, it is relatively easy to create data visualizations using Python.

Pandas provides many ways to access/read data. It allows the user to read the data from a local csv or excel file, pull the data from a relational database or data store, or read directly from a URL (when you have internet access). Since we are working with the Athena data store `iri_usda` in this course, we will demonstrate how to use pandas to read data from a data store. For examples to read data from a CSV file, refer to the pandas documentation Getting Data In/Out (pandas.pydata.org/pandas-docs/stable/10min.html#getting-data-in-out).

The function to run a SQL query and pull the data into a pandas dataframe (more to come) is `pd.read_sql()`. Just like running an SQL query in DBeaver, this function will ask for some information about the data store and the query you would like to run. Let's walk through the example below.

Establish a Connection to the Database

The first parameter is the connection to the database. To create a connection, we will use the `Pyathenajdbc` package and tell it which data store we want to connect to, just like in DBeaver. Additional details on creating a connection to the data store are provided in the Databases (01_1_Database_Connections.ipynb) notebook.

> In your own work, you can reach out to your IT team to help get connected to your database. They should provide you with connection parameters.

Database Connection

In [ ]:
```
conn = connect(s3_staging_dir = 's3://usda-iri-2019-queryresults/',
               region_name = 'us-gov-west-1',
               LogLevel = '0',
               workgroup = 'workgroup-iri_usda')
```

Formulate Data Query

create a query as a `string` object in Python

In [ ]:
```
# query to pull demographic data on 20 households in 2017 enrolled in th
e WIC plan
query = '''
SELECT *
FROM iri_usda.demo_all
WHERE wic_june = 1 and year = '2017'
LIMIT 20;
'''
```

> Note: When used together, the three quotation marks surrounding the query body is called a multi-line string. It is quite handy for writing SQL queries because the new line character will be considered part of the string, instead of breaking the string.

In [ ]:
```
# Now that we have defined a variable `query` in Python, we can call it
 in the code
print(query)
```

Here, we use the `LIMIT` statement for two reasons. First, `LIMIT` helps users avoid running into memory issues in Python, as the command controls the maximum amount of rows of the dataframe. Second, it will also speed up some queries for the same reason. Generally, when performing an exploratory data analysis using SQL commands, we recommend you use `LIMIT` to look at a small sample of the data rather than wasting time and potentially creating memory issues by looking at the entire dataset. Sometimes, it may also be advantageous to provide robust `WHERE` clauses that will naturally limit the size of the output, such as restricting the resulting dataset to a specific year, or `state` , in this case. For instance, if you were curious how a given variable within the demographic data changed by year, you could start by restricting the dataset to just 2012 and then systematically change the year until you had a full sense of the trend (or lack thereof) in the dataset instead of grouping by the year from the start.

> Note that `LIMIT` provides a simple way to get a "sample" of data. However, using `LIMIT` **does not provide a *random* sample**; it is just based on what is fastest for the database to return.

Pull Data from the Database

Now that we have the two parameters (Athena connection and query), we can pass them to the `pd.read_sql()` function to obtain the data.

In [ ]:
```
# here we pass the query and the connection to the pd.read_sql() functio
n
df = pd.read_sql(query, conn)
```

In [ ]:
```
# the first five rows of our 2017 WIC households pandas dataframe
df.head()
```

In [ ]:
```
# more information on df
df.info()
```

Analysis: Using Python and SQL

Let's recall our guiding questions:

What are WIC households' total food expenditures in 2017? What is the share of WIC purchases for these households?

To find the answers to these questions, we will need to combine the demographics data with another available dataset. We will start slow and explore the two datasets individually, and then work up to answering these questions. Our process will work as follows:

- Explore the available data tables
- Check out the demographics table and some distributions within it
- Define our mystery table
- Explore mystery table
- Combine datasets
- Answer questions

> Note: `demo_all` is a longitudinal data table, meaning that there may be multiple rows for one `panid`, or household, in the dataset if the household was tracked for multiple years.

What is in the Database?

As introduced in the Databases (./02_1_Databases.ipynb) notebook, there are a few different ways to connect and explore the data in the database.

Tables, and Columns in database

Let's pull the list of table names in the database and the list of columns in these tables to get more familiar with the `iri_usda` datasbase. You only have read permissions for `iri_usda`, meaning you cannot create tables to this database. There is another database, `iri_usda_2019_db`, which you can directly write, or create tables, to.

In [ ]:
```
# See all available tables
query = '''
SHOW tables IN iri_usda;
'''
pd.read_sql(query, conn)
```

In [ ]:
```
# check column names for a given table
query = '''
SHOW COLUMNS IN iri_usda.demo_all;
'''

pd.read_sql(query, conn)
```

Checkpoint 1: Explore The Tables

Use the following code cell to figure out which table could be one we will use later to answer our guiding questions.

Are there any tables you can see being useful in your own work? **Discuss with your group.**

In [ ]:
```
# check column names for a given table
query = '''
SHOW COLUMNS IN iri_usda.INSERT_TABLE
'''
```

Dive into Demographics Data

Looking at just Virginia households from the table `demo_all` within iri_usda, let's find the number of rows containing information from 2017. If you are confused about any of the variables and how they were encoded, please consult the data dictionary.

In [ ]:
```
# years for Virginia households in demo_all

query = '''
SELECT year
from iri_usda.demo_all
where state = 'VA'
'''

df = pd.read_sql(query, conn)

#first five rows of df
print(df.head())
```

In [ ]:
```
# count number of Virginia households from 2017 in df

print(len(df[df['year'] == '2017']))

print(len(df.query('year == "2017"')))
```

> Note the two ways to subset a `Pandas.DataFrame` :
>
> 1. Use the built-in `.query()` function
> 2. Create an array of `True` and `False` values with following format: `tables["column"] == 'desired entry'`

Let's check out the `wic_june` column. Ignore the `wic_jan` covariate and focus solely on `wic_june` to identify if households are enrolled in the WIC program. If `wic_june` is coded as a 1, then the household is enrolled in the WIC program. First, we'll breakdown the households.

> Note: The `projection61k` category contains survey weights for households the IRI believed had enough purchase entries so that they had a good grasp of the household's purchasing history. If `projection61k` is greater than zero, it means that there is enough purchasing data for that household to find general population estimates.

In [ ]:
```
# Count distinct households in dataset in 2017
query = '''
SELECT COUNT(DISTINCT(panid)) as household_count
FROM iri_usda.demo_all
WHERE year = '2017';
'''
pd.read_sql(query, conn)
```

In [ ]:
```
# Count of distinct households enrolled in the wic program in 2017
query = '''
SELECT COUNT(DISTINCT(panid)) as wic_count
FROM iri_usda.demo_all
WHERE wic_june = 1 and year = '2017';
'''
pd.read_sql(query, conn)
```

In [ ]:
```
# Percentage of WIC households out of those with enough purchasing data
 in 2017
query = '''
SELECT COUNT(DISTINCT(panid)) * 100.0 /
    (
    SELECT COUNT(DISTINCT(panid))
    FROM iri_usda.demo_all
    where projection61k > 0 and year = '2017'
    )
    as percentage_wic, COUNT(DISTINCT(panid)) as wic_total
FROM iri_usda.demo_all
where projection61k > 0 and wic_june = 1 and year = '2017'
'''
df = pd.read_sql(query, conn)

print(df)
```

Checkpoint 2: Take a Step Back

How does this sample compare to total WIC participation across the nation? **Discuss with your groups.**

We will have a more in-depth discussion about this sample and its representativeness of all households in the United States when we go through the Inference notebook.

Mystery Table Reveal

Necessary data:

- `iri_usda.demo_all` : individual household demographics data
- `iri_usda.trip_all` : individual household purchase data by item and date

Let's explore a few specific questions to better understand `trip_all` :

- How many distinct households are in `iri_usda.trip_all` ?
- How can you calculate the food expeditures for a single household at Walmart in 2017? What about the amount they've spent on WIC purchases? The `storename` value corresponding to Walmart is `3025` .
- What are the most popular methods of payment at Walmart in 2017?

In [ ]:
```
# number of households in the trips dataset
query = """
SELECT count(distinct panid) as purchase_panids
FROM iri_usda.trip_all;
"""

print(pd.read_sql(query, conn))
```

> Note: **Large tables** can take a long time to process on shared databases. The individual purchase table has more than 596.4 million records.

Household Expenditures

To calculate the amount of money a single household spent at Walmart in 2017, we need to know:

- How to calculate expenditures
- Which columns we need to focus on
- How to combine different SQL commands

First, let's focus on calculating household expenditures for ten households. Within `trip_all`, there are two relevant variables for this question: `dollarspaid`, which is the cost of the individual item, and `coupon`, which is the amount the price was reduced through the usage of a coupon.

In [ ]:
```
# example 10 household expenditures
query = """
SELECT panid, sum(dollarspaid) - sum(coupon) as total
FROM iri_usda.trip_all
GROUP BY panid
LIMIT 10;
"""
# print results
print(pd.read_sql(query, conn))
```

In [ ]:
```
# get entries of how much 10 households spent at Walmart in 2017
query = """
SELECT panid, sum(dollarspaid) - sum(coupon) as total
FROM iri_usda.trip_all
WHERE year = '2017' and storename = 3025
GROUP BY panid
LIMIT 10;
"""
# print results
print(pd.read_sql(query, conn))
```

In [ ]:
```
# example 10 household expenditures in 2017 at Walmart using WIC payment
query = """
SELECT panid, sum(dollarspaid) - sum(coupon) as total
FROM iri_usda.trip_all
WHERE year = '2017' and storename = 3025 and mop = '7'
GROUP by panid
LIMIT 10;
"""

# print results
print(pd.read_sql(query, conn))
```

Did you notice that some queries were faster than others? As we progress throughout the course, you will begin to run queries that may take much longer than five seconds to run. The more complicated the analysis is and the more data required to run the query affect the total runtime. But as discussed before, there are ways to easily cut down on runtime before creating exactly what you want using the entire data table. It is good practice to use `LIMIT` and/or `WHERE` clauses first to verify that the query is performing as intended before running a more extensive one.

In [ ]:
```
# proportion of money spent by each possible method of payment at Walmar
t in 2017 rounded to two decimal places
query = '''
SELECT mop, round(count(*) * 100.0 / (SELECT count(*)
    FROM iri_usda.trip_all WHERE year = '2017' and storename = 3025), 2)
as percentage,
    count(*) as count
FROM iri_usda.trip_all
WHERE year = '2017' and storename = 3025
GROUP BY mop
ORDER BY percentage DESC;
'''
# get results
df_mop = pd.read_sql(query, conn)
```

In [ ]:
```
df_mop
```

In [ ]:
```
# can see different columns in df_mop
df_mop.columns
```

Now, we have explored all the information we need to combine to find the answer to the two questions. To do so, we will have to join the datasets based on their values for `panid` .

Join Tables

To join `trip_all` with `demo_all`, we will need to match the two datasets by household, or `panid`. For this subset, we want to find the amount of money spent per WIC household with sufficient purchasing data. Before creating any sort of table, we would like to confirm that the join works.

In [ ]:
```
# joining data to get amount spent per household for those with sufficie
nt purchasing data in 2017

query = """
SELECT demo.panid, round(sum(trip.dollarspaid) - sum(trip.coupon), 2) as
total
FROM iri_usda.demo_all demo
LEFT JOIN iri_usda.trip_all trip
ON trip.panid = demo.panid
WHERE demo.wic_june = 1 and demo.projection61k > 0 and demo.year = '201
7' and trip.year = '2017'
GROUP BY demo.panid
LIMIT 10;
"""
# display result
pd.read_sql(query, conn)
```

In Athena, is impossible to create temporary tables, so we have to create regular tables and subsequently drop them if we are finished using them. In general, before creating a table, or temporary table, it is best practice to make sure the query runs as designed, and then assign it to a table. Here, a table is not necessary, but since we will be using the table for future calculations, we will create a table `panid_expense`, which contains each WIC-household `panid` with `projection61k > 0` for each year. From there, we can find the estimated amount of money spent in a given year by multiplying the amount spent by the survey weight.

We will be writing the `panid_expense` table to `iri_usda_2019_db`, which is the Athena database we all have write access to.

> Note: We will discuss the difference in the expenditure calculations without weights further in the Inference notebook.

In [ ]:
```
# see existing table list
table_list = pd.read_sql('show tables IN iri_usda_2019_db;', conn)
print(table_list)

# get a series of tab_name values
s = pd.Series(list(table_list['tab_name']))
```

In [ ]:
```
# create money spent per household table for WIC households with suffici
ent purchasing data in 2017
if('panid_expense' not in s.unique()):
    query = """
    CREATE table iri_usda_2019_db.panid_expense
    WITH (
    format = 'Parquet',
    parquet_compression = 'SNAPPY'
    )
    AS
    SELECT demo.panid, round(sum(trip.dollarspaid) - sum(trip.coupon),
 2) as total_wic_purchase, demo.projection61k
    FROM iri_usda.demo_all demo
    LEFT JOIN iri_usda.trip_all trip
    ON trip.panid = demo.panid
    WHERE demo.wic_june = 1 and demo.projection61k > 0 and trip."year" =
'2017' and demo."year" = '2017'
    GROUP BY demo.panid, demo.projection61k;
    """
    with conn.cursor() as cursor:
        cursor.execute(query)
```

In [ ]:
```
# check out iri_usda_2019_db.panid_expense
query = """
SELECT *
FROM iri_usda_2019_db.panid_expense
LIMIT 10;
"""
res = pd.read_sql(query, conn)
print(res)
```

In [ ]:
```
# we can get descriptive stats for panid_expense
query = '''
select total_wic_purchase, projection61k
from iri_usda_2019_db.panid_expense;
'''

df = pd.read_sql(query, conn)

df.describe(include='all')
```

Now that we have information for WIC household expenditures, we are quite close to the answer for the first question. How can we find an estimate for the total food expenditures for WIC households using the table we just created?

In [ ]:
```
# 2017 total food expenditures for WIC households
query = '''
SELECT sum(total_wic_purchase*projection61k) as part_1_answer
FROM iri_usda_2019_db.panid_expense;
'''

full_purchase_sum = pd.read_sql(query, conn)
print('WIC households spent approximately ${:.2f} in 2017.'.format(full_
purchase_sum['part_1_answer'][0]))
```

> We can parameterize Python `string` objects - using the built-in `.format()` function. We will use various formulations in the program notebooks (e.g. when building queries). Here are some other examples:

1. Empty brackets to insert the variable in the string; when there is more than one set of brackets Python will insert variables in the order they are listed.
2. Brackets with formatting can be used to make print statements more readable (eg `'text with formatted number with comma and 1-digit decimal {:,.1f}'.format(number_value)` will print `123,456.7` instead of `123456.7123401`).
3. Named brackets to use the same variables multiple times in a text block (we use this in more compicated queries like when creating "labels" and "features" for machine learning models).

What share do WIC purchases make up for WIC households?

By now, you should have a grasp of the demographic and purchasing datasets and also understand that we will need incorporate the `mop` variable as well.

Our approach is as follows: We are going to find two separate values, one of estimated total purchase amount for WIC households (already created), and one of amount of estimated purchase amount using the WIC program for WIC households. From there, we can divide the two values to find the answer.

In [ ]:
```
# reminder about part 1 calculated above: see answer to first question a
gain
query = '''
SELECT sum(total_wic_purchase*projection61k) as part_1_answer
FROM iri_usda_2019_db.panid_expense;
'''

full_purchase_sum = pd.read_sql(query, conn)
print(full_purchase_sum)
```

First, we will select all of the households that were enrolled in the WIC program in 2017. After that, we will utilize a list comprehension to subset for trips just for these households in 2017 to extract their purchase data.

In [ ]:
```
# list of panids from demographic data that are in wic program from 2017
query = '''
SELECT DISTINCT panid
FROM iri_usda.demo_all
WHERE "year" = '2017' and wic_june = 1 and projection61k > 0;
'''

wic_hh = pd.read_sql(query, conn)
```

In [ ]:
```
# convert to python string of strings
wic_hh_sql = ','.join(["'"+id+"'" for id in wic_hh['panid'].values])
```

This line of code may look complicated, so let's break it down step by step:

1. **`... for id in wic_hh['panid'].values ...`** - Loop through every element `id` in the list `wic_hh['panid'].values`
2. **`... "'"+id+"'" ...`** - String concatenation to make each `id` a string
3. **`','.join( ... )`** - Join each of these `id` strings into one string (a string of strings) with a comma separator

Additional Note: The formulation `[<action> for <item> in <iterable>]` is known as "list comprehension".

The same code can be duplicated in a classic `for` loop as well.

In [ ]:
```
for id in wic_hh['panid'].values:
    print("'"+id+"'")
```

In [ ]:
```
# total dollars spent using WIC purchases for WIC customers in 2017 esti
mate
query = '''
select round(sum(visit.dollarspaid * demo.projection61k) - sum(visit.cou
pon * demo.projection61k), 2) as total
from iri_usda.trip_all visit
left join iri_usda.demo_all demo
on visit.panid = demo.panid
where visit.panid IN (
    {}
    )
and visit.mop = '7' and demo."year" = '2017' and visit."year" = '2017'
'''.format(wic_hh_sql)


wic_purchase_sum = pd.read_sql(query, conn)
```

In [ ]:
```
wic_purchase_sum
```

Checkpoint 3: Final Step

Calculate the estimated percentage of WIC purchases for WIC households.

Note: There's more than one way to do this.

In [ ]:
```
# find percentage of WIC purchases for WIC households
```

Moving Forward

Based off of the guiding questions in these notebooks, we will extend these analyses a bit further in the sample project notebooks. The first sample project will build off of the analyses created in this notebook.

Additionally, the data visualization notebook will contain a visualization of the average amount spent on a specific product across different family sizes per household member.