Irrigation Data Visualization and Analysis Tool

Kirin Mackey

December 17, 2024

1 Project Description, Background, and Motivation

Irrigation is an essential device in the U.S. agricultural industry, as it improves efficiency of farms and offers a way for states that have a dry and arid climate to sustain themselves. According to the 2017 Census of Agriculture, 58 million acres of cropland were irrigated, and farms using irrigation supported 54 percent of the total crop sales in the U.S. (Hrozencik, 2023). Irrigation has many components, such as energy, facilities and equipment, labor, practices, water, pumps, and wells. All of these components have further details, which can reveal the negative effects of irrigation. For instance, acres irrigated with groundwater in Oklahoma could indicate the decline of the Ogallala Aquifer, which accounts for one fourth of the water supply used for agricultural production in the U.S. (Hanrahan, 2024) and supplies more than 20 billion dollars worth of food (Little, 2009). On the other hand, these details can reveal sustainable practices and indicate future irrigation use, such as the amount of land irrigated with recycled water or the amount of land equipped for irrigation in a certain state. Learning about these details and finding statistics about them is an arduous task that involves searching through multiple research reports, especially if the user wants to look at a particular state or year.

To learn about these revealing details in an effective manner, this project makes a tool using Dash in which a user can either get visualizations or tables about them using data at the state level from the United States Department of Agriculture (USDA). A user can specify an individual or multiple states, the specific data they want to visualize or analyze, and what years the data to be visualized or analyzed reflects through the use of dropdown lists and buttons. If they want a visualization, they will be given the option to choose a line graph or bar chart, each of which will affect the amount and which type of items the user can visualize. The user will also be prompted to choose which type of statistic they want displayed if applicable, such as minimum, maximum, average, and sum. The user will also have the option to save the visualization the tool creates as a png file, and save the data table as a csv file. The tool can be used to aid in creating research reports, serve as reference material for government officials, and be used in classroom settings for students studying environmental science and its associated public policies and economics.

2 Data Description

The data the tool, or Dash app, presents to the user was collected by the USDA and National Agricultural Statistics Service (NASS) through the Census of Agriculture. The census is performed every five years and is first executed by identifying sources of agriculture that meet the definition of the farm, which the USDA (2022) states is a "place from which $1,000 or more of agricultural products were produced and sold, or normally would have been sold, during the census year." The identification process is done by the NASS, which has an ongoing list of sources that meet the definition of a farm. This list is contributed to by state governments, producer associations, seed growers, veterinarians, pesticide applicators, and community organizations. There are also field offices of the NASS in which its staff finds potential places that can be considered farms to collect data from. This ongoing list is called the census mailing list, and the sources listed are sent a packet of forms to fill out through the mail. They can also fill it out online through invitation (USDA, 2022). The form accounts for many aspects of agriculture, asking the participants for information about the crops or livestock they grow or raise, how they use the land, the practices they employ, their expenses, their incomes, how they use labor, whether they use renewable energy, their equipment, and personal characteristics of the farmers. One section of the form asks respondents to fill out information about irrigation.

Based on the information given by respondents, the NASS then processes the data, where the data is checked on whether the source meets the criteria for a farm. Data analysts then determine whether there are outliers in the data, and perform classification methods on the data to account for undercoverage or possible previous misclassification. Once these processes are fully complete, the NASS compiles census estimates for all the categories questioned about in the census form for various geographic levels (USDA, 2022), to which then can be accessed through the NASS's searchable database named Quick Stats (found at https://quickstats.nass.usda.gov/). The data this project uses was found by searching this database for irrigation data relating to the state level, as it was deemed an appropriate scale for the scope of this project. The raw data obtained by this search can be found in the data folder in the GitHub repository for this project under the name Irrigation_Data.csv, and is automatically provided to the user once they clone the repository.

After preprocessing the raw data, the columns of the resulting data table are listed and described below, using the Quick Stats parameter definitions as a guide:

Column Name	Description
Year	The numeric year of the data
State	Full state name

State ANSI	American National Standards Institute (ANSI) standard 2-digit state codes.
Commodity	The primary subject of interest related to irrigation (ENERGY, FACILITIES & EQUIPMENT, LABOR, PRACTICES, WATER, WELLS, PUMPS)
Data Item	Describes an aspect of the commodity measured Describes an action taken upon the commodity, how it is used, and the unit it is measured with (thus is specific to a domain on a similar level as domain category) Example: For the WATER commodity, a data item is RECYCLED, IN THE OPEN - ACRES IRRIGATED. This indicates the item in this row, belonging to the commodity WATER, is being measured by acres irrigated (units) in the open(way item is used) with recycled (action taken upon commodity) water.
Domain	A characteristic of the operations (farms/ranches/producers) that relate to a particular commodity, such as EXPENSE for the commodity ENERGY. The domain TOTAL will have no further categories for the domain category field.
Domain Category	Categories within a domain (when domain = EXPENSE, domain categories include $1,000 TO $1,999, $2,000 TO $4,999, 5,000 TO 9,999 $ and so on).
Value	Data Value

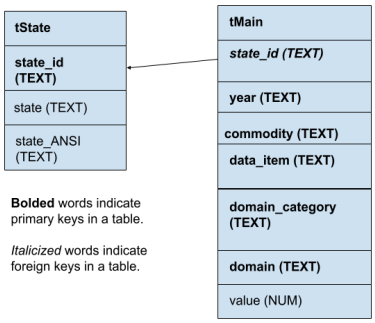
All these pieces of data were found to be the most relevant because they succinctly define an aspect of irrigation and the multiple nuances hidden within it. Since the nuances are layered, such as commodity, domain, data item, domain category, they provide ways in which a user can filter through different categories of data. For instance, a user can filter by commodity to find certain domains, and filter by domain to find data items that fall under it, and then filter by data item and domain to find available domain categories. These then hold values attached that

describe the data item for a particular state and year. State, state ANSI, and year are included so that a user can compare data items and/or domain categories across time and location. From these comparisons, a user can find trends about irrigation. For instance, a user can compare the amount of farms pertaining to different expense brackets in which expenses are used to deepen wells. This can reveal how farms allocate their funds for potential new farmers or if the selected state(s) are struggling to find water for government agricultural officials. It can also suggest depletion of groundwater sources and worsening of soil conditions for students concerned about the effects of irrigation.

For further information on location, data provided by the CDC is used in this project to provide abbreviated names for each state. There are only the full names for states in Irrigation_Data.csv, but they are not useful in creating effective visualizations. The data with abbreviated state name data was originally used by the CDC in teaching users how to make data visualizations with maps. The data was originally found at https://www.cdc.gov/wcms/4.0/cdc-wp/data-presentation/data-map.html, but is in the data folder in the GitHub repository for this project under the name data-map-state-abbreviations.csv. It is automatically provided to the user once they clone the repository. It should be noted the data includes both states and territories of the United States. The columns of the data table and their descriptions are provided in the table below:

Name	Name of state or US territory
Abbreviated	2 letter abbreviation of state or US territory

All of the data specified from the two .csv files is compiled into a relational database with two tables tState and tMain. From the column in Irrigation_Data.csv, year matches Year, commodity matches Commodity, data_item matches Data Item, domain_category matches Domain Category, domain matches Domain, state matches State, state_ANSI matches State ANSI, and value matches Value. From the columns in the data provided by the CDC, state matches the Name, and state_id matches Abbreviated. An ERD depicting the relationship between the two tables is provided below:



All items are text items stored as strings with the exception of those in the value field, which is of the numeric data type because there are values that have decimals in the data provided by the USDA. The value field is also what the aggregate statistic functions chosen by the user are performed over.

The table tMain holds all data related to the irrigation data collected by the USDA. Notably, 6 out 7 columns are primary keys. This is because after data exploration, it was found that each entry in value is specific to a unique combination of state_id, year, commodity, data_item, domain_category, and domain. Moreover, each entry in data_item is specific to a unique combination of domain and commodity, while each domain_category is specific to a unique combination of domain and data_item, and therefore commodity. To keep within the scope of the project at the time of data collection, it was deemed appropriate to address the dependencies by keeping the structure of 6 primary keys.

The table tState holds all data related to the name of the 50 US States. The primary key is state_id, as it is a succinct and accessible identifier of each US state. Because of the field's accessibility, it serves as the identifier for states in tMain, where it is a foreign key. The entries in state_id are used in the final tool to serve as state identification in the visualizations/data tables presented to the user due to their conciseness. The data in state and state_ANSI is kept for future work, which is elaborated upon in the Future Work section.

3 Methods

The Dash app presents selections to a user that filter the irrigation data described in the Data Description section of this proposal. The selections follow in the order of type of visualization/format of data table desired, state(s), commodity, domain, data item, domain category(ies), year(s), and type of statistic desired to be shown. There is a special case where the user picks TOTAL as the domain. Here, since there are no domain categories to filter the irrigation data further, the user is given the choice to visualize one data item (the one they chose first), or multiple, where each additional data item also has a domain of TOTAL and uses the same units as the one initially selected. After this the user is presented years to select from.

In general, the tool determines what field is being analyzed by looking at the amount of items chosen for the year, state, data item, and domain category fields. For example, if multiple domain categories have been chosen with multiple years and multiple states, the bar plot or line graph will compare the different domain categories to each other, with the values for each computed by using data from the multiple states and multiple years specified. However, there are special cases that require more user input.

One of the special cases is triggered in the case the user wants a bar plot, has specified only one data item (domain is set to TOTAL) and possibly one domain category (domain isn't set to TOTAL), and then either multiple years and multiple states, or one year and one state. The tool needs to know what the user wants on the x-axis, states or years, and thus what their chosen statistic is representative of. It then will ask the user after they specify their desired statistic this question.

The other special case is triggered in the case the user wants a line graph, has specified multiple states, has specified only one data item (domain is set to TOTAL), and possibly one domain category (domain isn't set to TOTAL). The tool needs to know whether the user wants multiple lines on the graph, each pertaining to one state, or one line that is representative of all of them. This question is asked after the user chooses their desired statistic. If the user chooses multiple lines, the values displaying the statistic's change over time on each line on the graph are computed using the values for the isolated data item/domain category for a particular state chosen. If the user chooses one line, the values displaying the statistic's change over time are computed using the values for the isolated data item/domain category for all states selected by the user.

After these special case questions, if necessary, are asked, 3 buttons are displayed to the user. They allow the user to obtain a graph of their data specifications (driven by plotly), save the figure as a .png, and obtain a data table representation of their data specifications (saved as a .csv file). The save figure button is disabled until the generate graph button is disabled, and all three buttons are disabled once you click them. If an earlier selection is changed by the user, the outputs of the selections after, including the existence of the buttons can be reset if the change proves certain later selections invalid. If the later conditions prove to still be valid, the three buttons will reset (with the save figure button initially disabled) since they will display slightly different data.

For a full examination of the Dash app, it must first be known that is built upon 6 .py files: main_dash.py presented to the user in the main directory of the project's GitHub repository, and Irr_DB.py, irrigation_base.py, data_table.py, visualization.py, __init__.py found in the src directory given to the user.

It should be noted that the __init__.py file is empty. It is created so that the other files in the src directory can be treated as modules within a package, that being src. This makes it so that across all the .py files, they can import classes and/or functions from each other by calling src.<insert_file_name_holding_class_or_function>. Without doing this, ModuleNotFoundErrors will be created.

The file main_dash.py initially creates the Dash app and sets the theme to a dash bootstrap theme. An html.Div item holds all of the components of the app by taking in a list of children, These children are then created and ordered by the layout function. The order of them added to the list is the order in which the components are added to the webpage. The layout function defines all of the html, dash, and dash bootstrap components. Many of these items have an empty default value, which are then set by the output of callbacks.

The first item added to the webpage (list of children) is the title, welcome message, additional notes, and warnings the user should know before traversing through the tool. After this, a pair of radio buttons and a label are added to ask the user what type of visualization they want (a bar plot or line graph). These buttons and label always appear since they are not dependent on any past selection. The value the user picks is then stored as a string (either 'Line Graph' or 'Bar Plot').

The next item to be added to the webpage (list of children) is a checklist detailing the states a user can choose from to reflect the data they want visualized/analyzed. The checklist must be organized as a list of dictionaries (each key and value is a string), where for each item (element in the list) there is a dictionary holding the label of the item and its associated value. For the states checklist, these labels and values are the same since they are state abbreviations, equivalent to the state_id field in tMain. To access these values, Irr_DB().get_states() must be called, or the get_states() function within the Irr_DB class. If it is the first time the user is using the Dash app, the irrigation database will be created in the data folder when this function is called.

The irrigation database is defined as the Irr_DB class located in Irr_DB.py. The Irr_DB class has inheritance with the base database class DB located in irrigation_base.py. In the constructor for Irr_DB, the constructor for the parent class DB is called, with a path specified as 'data/irrigation.db' passed in and a boolean create passed in as True. The constructor of the DB class uses the os module to check whether the path exists or not. If it does not and create = True, it sets up a connection for the database and closes it. It also sets up an attribute self.exists as False. This is so that in the constructor for the Irr_DB class, it then calls the functions build_tables() and load_tables() from the base DB class in irrigation_base.py. When Irr_DB() is called in main_dash.py any time after this , the path already exists and self.exists = True, where then build_tables() and load_tables() are not called.

When the irrigation database is being created by build_tables(), it is checked whether the tables tMain and tState exist. If they do, they are dropped using cursors. Otherwise, empty tables for them are created through CREATE TABLE sql queries and using cursors to execute the queries. Their primary keys, foreign keys, other columns, and data types for each column are described by the ERD in the Data Description section of this project report. The function

load_data() is then called, which then immediately calls another function in the Irr_DB class, prep_data(). The function prep_data() first reads the data in Irrigation_Data.csv as a pandas DataFrames. It cleans the data, and drops all the unnecessary columns initially held in Irrigation_Data.csv, such as Week Ending, Geo Level, Ag District, Ag District Code, County, County ANSI, Zip Code, Region, watershed_code, Watershed, CV (%). These fields were either empty upon obtaining the irrigation data from the USDA, or had no relevant information to the project. All rows not relating to year-based data are then dropped as well by finding the indexes in which rows don't have year specified as their time periods of data collection. A similar process is done when removing rows that have (Z) or (H) as the entry in the Value column. Additional commas are then removed from all entries in the Value column, and the Year, State ANSI, and Value columns are set to their proper data types (string, string, and float respectively).

Many columns of Irrigation_Data.csv are then cleaned to remove redundant information. The Domain Category column is cleaned by iterating through the rows with iterrows(), removing redundant information about the associated domains and getting rid of semi-colons by using the string method split and splitting on the commas. For cleaning the Data Item column, each commodity has a different pattern of displaying redundant information for their corresponding data items. Rows for the Data Item column are iterated through based on the commodity value. In getting rid of redundant information and additional spaces, each row is often split on the commodity name. After every split, the essential information is joined together and added together again to form the new entry for that row. After this process is performed for every commodity, the pandas dataframes to be converted to relational tables are set up.

Next, the prep_data() function takes a subset of state data from the cleaned irrigation data, State and State ANSI. It performs drop_duplicates to only get the unique pairings of the values in these columns. The CDC state data is read as a pandas dataframe, and all the state names held under the Name column are converted to all uppercase to match the entries in the data collected by the USDA. The columns are renamed state (data is full state name) and state_id (data is abbreviated name of corresponding state). The pandas.merge function is done on both the subset of state data and cdc state data, where the merge is done on the state field. This by default is an inner merge. Because the only entries common in both are 50 states, any US territory information in the CDC data is filtered out. A new dataframe is created called tState, holding data for each state's abbreviation, full name, and ANSI code. The duplicate combinations the entries in the State, Year, Commodity, Data Item, Domain, Domain Category, and Value are filtered out of the newly cleaned irrigation data, and its columns are renamed to match the columns specified for tMain the ERD in the Data Description section of this project. These new dataframes act as values to keys 'tState' and 'tMain' respectively and are returned to the load_data() function.

In the load_data() function, the preprocessed data created in prep_data() is loaded into the appropriate relational tables (tMain or tState) of the database using sql queries requiring inputs and by calling load_table(). The load_table() function takes in a sql query stored as string named sql, uses the to_dict function in pandas to convert the pandas DataFrame passed in as data into a list of dictionaries, and inserts the data row by row into the table specified by the sql query. After this is done for both tMain and tState, the irrigation database is fully constructed and found in the data folder given to the user upon cloning the GitHub repository as irrigation.db.

Back in Irr_DB().get_states(), the get_states() function from the Irr_DB class is called. It queries tMain for distinct states using the function run_query(sql, params) in the DB class, where sql is the string detailing the sql query and params is any parameters needed to be defined by user (in the form of a dictionary with each key a string and each value is a list of strings). The database is connected to in run_query, and pd.read_sql is performed. The connection to the database is closed and the results of the query are returned in the form of pandas dataframe. To access these results, Irr_DB.get_state() calls.values.flatten.to_list() to get the results as a one dimensional list of strings. In main_dash.py when creating the states checklist, this list is iterated through to create each dictionary required to make each item in the checklist.

The amount of items a user can select in this section is 5. This limitation is done in the callback section in main_dash.py with the update_multi_options(value, viz_type) function. It takes in a list of strings detailing the states already chosen by the user as value, and the visualization type (passed in as string viz_type) the user has specified. If the visualization type hasn't been chosen yet, the state section is not displayed. If the type has been chosen, the function limits the amount of states the user can choose to 5 by resetting the options for 'state-cl' (the id for the state checklist) to have disabled items if 5 have already been clicked (adjusts the disabled property for each item through dictionary comprehension). It returns the options for 'state-cl' as a list of dictionaries of strings (both as keys and values). For whether or not the state section is displayed, the function returns its style as a dictionary of strings twice (once for the header and the other for actual checklist), where the key is 'style' and the value is either 'inherit' (displays as a checklist) or 'none'(doesn't display).

After at least one state is checked in the Dash app, the select commodity section will appear to the user. It is presented through a label and dropdown list, items of which are added to the children list in the function layout(). In layout(), the initial options available to a user is an empty list, but in the callback section with the display_coms(state_id) they are able to be presented to the user. The function display_coms takes in the user's choice of state(s) as a list of strings state_id. If no states have been chosen, state_id is a list with an empty string, and the commodity dropdown and header are not shown by setting the value of ‘none’ to a ‘style’ key for two dictionaries returned. If the user has done a selection of states, meaning the list state_id has a length, the commodity dropdown and header are then displayed to the user by setting the value

for a ‘style’ key in a dictionary to ‘block’. The options are the results from calling Irr_DB().get_commodity(), where get_commodity() is in the Irr_DB class found in Irr_DB.py.

At this time, when Irr_DB().get_commodity() gets called, irrigation.db already exists, so the build_tables() and load_tables() functions are not called. With get_commodity(), it queries tMain with run_query(sql, params) for distinct commodities (energy, facilities & equipment, labor, practices, pumps, water, and wells). It does not change based on previous user selections (state_id), so no params argument needs to be passed in. It then returns a list of the available commodities a user can choose from, where each element is a string. This list is returned to the display_coms function in main_dash.py, to which then each element is an option in the dropdown for commodity (identified by 'com-dd'). When a user selects a commodity from the dropdown, the value stored is a string.

After a commodity has been chosen, the select domain section will appear to the user. It is presented through a label and dropdown list, items of which are added to the children list in the function layout(). In layout(), the initial options available to a user is an empty list. They are set in the callback section with the update_doms(state_id,commodity) function. The update_doms function takes in values for state(s) as state_id (a list of strings) that were chosen by the user in a checklist and takes in the commodity chosen via dropdown (stored as a string called commodity). This is because domain options are dependent on these previous selections. It queries the irrigation database, using the Irr_DB().get_domains(comm_params) function. It passes a dictionary comm_params with strings as the keys (‘state_id’ and ‘commodity’) and their corresponding values as a list of strings to get a list of valid domains.

The get_domains(comm_params) function is within the Irr_DB class. It takes in a dictionary comm_params, in which its keys are strings and each value is a list of strings (only includes state_id and commodity at this step). It sets a string that utilizes json and json trees to query the database with run.query(sql, params). The json structure makes it so that a list with multiple items can be passed in as a parameter to a sql query. Therefore the query, executed by run_query(sql, params), returns valid domains where the state field is the same as any item in the list of states, and where the commodity is equal to that specified by the value passed in the with key ‘commodity’. It returns a list of strings with each element a valid domain to the update_doms function in main_dash.py. In the update_doms function, it checks whether results of the query to the database have any length. If there are items in the results (called vals in this function), the dropdown to choose a domain is presented to the user along with its label since style is set to block in a dictionary. If no results were returned, it means that states and/or commodity have not been chosen by the user. In this case the style is set to none, so the dropdown to choose a domain is not displayed. If the domain dropdown is presented, the user’s selection will be stored as a string.

The next available selection is for data item, which is presented by a dropdown and label that are added to the children of the list built in the layout() function in main_dash.py. The options and style of these items are not set in layout(), but rather in the callback section with the function update_dts(state_id, commodity, domain). The function takes in the previous selections by the user: state(s) with state_id (a list of strings), commodity chosen with commodity (a string), domain chosen with domain (a string). This is because data item selection is dependent on state(s), commodity, and domain specified by the user. It queries the irrigation database, using the get_data_items(dt_params) function in the Irr_DB class to get valid data items to choose from. It passes in dt_params as a dictionary with its keys being 'state_id', 'commodity' and 'domain'. Each value for these keys is a list of strings (in the case of state_id) or a list with 1 string (in the case of 'commodity' and 'domain'). This is done to employ the json trees when passing lists with multiple elements, such as the one held with the key 'state_id' as parameters in a sql query. In get_data_items(dt_params), it uses the dictionary dt_params passed in and queries the irrigation database using run_query(sql, params). If any of the parameters needed to construct dt_params, (state(s), commodity, or domain) have not been selected by the user, the query does not return any valid results to update_dts in main_dash.py. The dropdown to choose data items does not display as a result. If there are results, they are presented as options for the user to choose from in the dropdown that is displayed because its style is set to block rather than none.

After the data item has been selected, the branching off point with regards to whether domain =TOTAL begins to happen. The case in which domain = TOTAL will first be discussed.

If TOTAL was selected by the user to be the domain, after a data item selection, radio buttons and a corresponding label may appear to the user asking whether they want to visualize/analyze multiple data items, or just one data item. In the layout() function in main_dash.py the options stating the text for the buttons are empty. They are set in the callback section with the function ask_mult_dt(domain, data_item). To know whether to ask the question, it takes in the user selected domain as domain (a string), and user selected data item as data_item(a string). If the data item selection exists, and if domain = TOTAL, the radio button item mult-dt-r is displayed with the options 'Multiple Data Items' and 'One Data Item', which are returned in a list. Otherwise, this question does not appear. The appearance is controlled by two dictionaries returned (each key and value is string), both of which have 'style' as the key and 'inherit' as the value if the question and buttons are shown, or 'none' as the value if the buttons are not shown. The 'inherit' keyword indicates the style is inherited from the parent, which in this case is radio buttons.

If the domain is selected to be TOTAL, and the user selects 'Multiple Data Items' when asked whether they want to visualize/analyze additional data items, they will be presented with a checklist where each item is a valid additional data item. These additional data items also have TOTAL as their domain, and the same units as the initial data item selected by the user. The user

is also limited to choose 4 items in this checklist. This checklist is added to the list of children in the layout() function in main_dash.py, but each item in the checklist is determined by the function update_mult_dts_items(mult_dt_q, state_id,commodity, domain, data_item, init_vals) located in the callback section of main_dash.py. It takes in the answer to the previous question (mult_dt_q, a string), and all values for previous selections made according to the type of their values (dependent on what type of item they are). This makes it so state selection is a list of strings state_id, commodity is a string commodity, domain is a string domain, and initial data item is a string data_item. These are needed because the additional data item field is dependent on all of these selections. To find the data items that can be compared with the initial data item chosen, the get_domain_categories(dc_params, mult_dt_q) function defined with the Irr_DB class is called.

The get_domain_categories(dc_params, mult_dt_q) function uses a dictionary dc_params passed in, in which its keys are strings and each value is a list of strings (only includes state_id, commodity, domain, and data_item at this step). A string mult_dt_q is also provided, which holds the user's answer to the question of whether they wanted to visualize multiple or one data item. The function checks whether the user specified domain as TOTAL. If they did, the number_dt_question(mult_dt_q), also within the Irr_DB class is called. This function acts as an encoder by using a dictionary. To adhere to the condition in the if statement in the block, handing the domain = TOTAL case in get_domain_categories, it returns an encoded version of the user's previous choice to 'multiple' or 'one'. In the get_domain_categories if the result of number_dt_question is ‘one’ the function returns nothing (this indicates the user chose ‘One Data Item’ when asked about the number of additional data items. If the encoder returns with the value ‘multiple’, the intermediate_domain_categories(idc_params) function, also within the Irr_DB class, is used to query the database.

The intermediate_domain_categories(idc_params) function takes in the dictionary passed into get_domain_ategoires as dc_params as its argument idc_params. It looks at the data item in idc params, finds its units, and sets the string that utilizes json and json trees to query the database with run.query(sql, params). It does so by adding another key to the dictionary called ‘more_data_items’ and sets the value to the unit used by the initially selected data item. The sql query then queries the database for data items that adhere to the same previously selected data specifications, end the same way as the string described in the key ‘more_data_items,’ and are not equal to the data item already selected. If there are no results to this query, the checklist for additional data items does not display (the dictionary {‘style’:’none’} is returned for the additional data item checklist style). If there are results, they are returned to get_domain_categories, and then the update_mult_dts_items function in main_dash.py as a list of strings.

The function also needs the values already chosen by the user in this checklist (a list of strings called init_vals) to limit the amount of items a user can select in the checklist to 4. This is

because the callback remembers past selections for this item (init_vals, a list of strings), even if they were for different previous data specifications (state, commodity, domain, data item). To combat this, the function finds items already selected that match the results of additional data items returned by the get_domain_categories function and ensures they match the units of the initial data item selected as well. Then it determines which items in the newly returned checklist of additional valid data items are disabled based upon the amount of valid data item values already selected by the user. To do this it employs dictionary comprehension to set the disabled property of these items.

In the case where the user didn't choose the domain as TOTAL, and therefore only one data item, a different checklist displaying possible domain categories may be shown to the user if all previous selections have been made. This condition is checked and the checklist and corresponding label are not shown by setting their to none. The checklist for domain categories is added to the list of children in the layout() function in main_dash.py, and the values of the items is set in the function update_dc(state_id, commodity, domain, data_item, init_vals) in the callback section of main_dash.py. A list of valid domain categories is found using the arguments state_id, commodity, domain, and data_item as the values of a dictionary with keys matching their variable names. Each of these values is converted to a list of strings if not already one. This dictionary is passed as dc_params to the function get_domain_categories(dc_params, mult_dt_q) function in the Irr_DB class, where mul_dt_q is None since the question for displaying multiple data items was not triggered. This function sets the appropriate query utilizing json trees so that lists with multiple items can be parameters. The results of this query executed by run_query are returned as a list of strings to the update_dc function in main_dash.py. However, since callback remembers past selections of domain categories for different previous specifications, the function determines the already valid domain categories by comparing the two lists (init_vals and the results from the sql query). Once the amount of valid domain categories reaches 5, the rest of the items in the checklist are disabled by setting their disabled property to True in dictionary comprehension.

After at least one item is chosen in the checklist relating to either additional data items or domain categories, the user will be presented with another checklist to select the years they want their visualization/final data table to reflect. This checklist is added to the list of children created in layout(), but its options and appearance is determined by the function update_years(mult_dt_q, state_id, commodity, domain, data_item, add_data_item, domain_category, init_vals).

The function constructs the checklist of valid years users can choose for their data specified above to reflect. The valid years are dependent on all previous selections, so accounts for cases in which the user chose domain = TOTAL (includes choice of multiple data items or one data item), and when they didn't need to answer the question. It then takes in the answer to multiple data item as a string mult_dt_q if applicable, state(s) as a list of strings state_id,

commodity as a string commodity, domain as a string domain, data item as a string data_item, possible additional data items as a list of strings add_data_item, and possible domain categories as a list of strings domain_category. If any essential specifications(state(s), commodity, domain, data item) haven't been filled out, the style of the checklist is set to None and it doesn't display. Because of the callback remembering past year selections stored in init_vals (list of strings), a query to the irrigation database about valid years pertaining to the most current data specifications is needed. To do this, either valid additional data items or valid domain categories are checked since valid years are dependent on them in addition to all previous data specifications. This is performed by using conditional statements that would trigger these cases (dependent on whether domain_category was passed in and whether mult_dt_q has an answer). There is then a checking procedure similar to that described previously for the individual sections for additional data items and domain categories. Once valid additional data items or domain categories are found, a dictionary is constructed to query the irrigation database for valid years. If in the instance the user chose domain = TOTAL and 'One Data Item' when asked if wanting to display multiple data items, the dictionary constructed contains all the earlier data specifications (state_id, commodity, domain, data_item as the keys in string format, the each value being a list of strings]. This dictionary is passed as year_params to the function get_years(year_params) in the Irr_DB class.

In the get_years(year_params) function, the previously used method of using jsom trees to hold lists that act as parameters for sql queries is effective. Because of json tree structure and IN() sql operator that essentially acts as the OR operator , it was found that run_sql_query(sql, params), where params=year_params, would return years that would be valid for a state, even if it wasn't valid for another The same can be said about keys that have values that are lists with more than a length of one (possibly data item and domain category depending on user selection).

To address this issue, for each list (value associated with a key in the dictionary year_params) with multiple items, the function finds valid years for each entry in those lists, and then finds years common to all those entries. This is done in particular for the list detailing the states chosen by the user,and the cases where the user selected more than one domain category or more than one data item. To find the valid years for values of the dictionary that are lists with more than 1 element in year_params, the function each_choice_year(key_name, year_params) located in the Irr_DB.py class, is called.

The function each_choice_year(key_name, year_params) gets valid years common to all entries in a list of strings, which is stored as value in the dictionary year_params. To look at an individual value (list) in year_params, the function takes in a key_name as string. For each entry in the year_params[key_name] list, the function finds the valid years by using the json method and run_query(sql, params) described previously. Then each_choice_year determines the years that are common to all of those entries by utilizing numpy.unique() and counting the number of

times each unique value appears compared to the total times it appears in a list. This list holds all the years returned for each label within year_params[key_name]. The function then returns the list of valid years for a particular data category to the get_years function.

Once back in the get-year function and finding years common to all entries in each of those lists, the get_years function compares the lists of valid years to each other if needed (when more than one data item or domain category selected), and finds the common years between those. If needed to compare multiple "valid year" lists to each other, the function uses set intersection to find the common years between all lists. If there aren't multiple "valid year" lists to compare to each other, it still looks at the valid years common to all states specified in year_params['state_id]. It sorts the final "valid year" list in ascending order, as once becoming a set the order of list elements does not matter. This is done to match up the labels of years to the final results, as SQL presents results by default in ascending order. To account for the possibility of more than 5 years being deemed valid, the process described for other checklist items and disabling certain items within it occurs here as well. When selected by a user, the valid years are stored as a list of strings.

After valid years are selected, the user must choose the statistic they want applied to the data they have specified. Radio buttons are added to the list of children developed in the layout() function in main_dash.py, and their appearance and values are determined with the function ask_stat(mult_dt_q, state_id, commodity,year, domain,data_item, add_data_item, domain_category). The existence of the buttons is dependent on all previous specifications possibly made, so takes in the answer to multiple data item question as a string mult_dt_q if applicable, state(s) as a list of strings state_id, commodity as a string commodity, domain as a string domain, data item as a string data_item, possible additional data items as a list of strings add_data_item, and possible domain categories as a list of strings domain_category. The function checks if all specifications are made for each possible case of domain=TOTAL, whether or not additional data items were specified, and the question of displaying whether the question of displaying multiple data items or not has been answered. It also checks if all specifications are made when domain isn't equal to TOTAL (whether domain categories have been specified). Because of this and the callback remembering previous selections discussed above, the function checks the validity of either additional data items, and in that whether the multiple data item question has been answered, or checks the validity of domain categories selected. In each case it constructs a valid query to the database to check for years possible with the get_years function. The ask_stat function also checks the validity of years chosen by the user, as the callback remembers year selections for different data specifications as well. If all required conditions have been met and there are valid years selected by the user, the radio buttons with the options Sum, Minimum, Maximum, Average, are presented, where their style is set to inherit rather than none. The user’s choice of statistic is stored as a string.

The special cases specific to bar plots and line graphs previously described occur after the user selects their desired statistic.

For bar plots, a pair of additional radio buttons and a label for them are displayed, asking the user whether they want state or years on the x axis. To determine whether these radio buttons are triggered and are added to the list of children built in the layout() function, the function ask_barplot_xax(viz_type,state_id, commodity, domain, data_item, mult_dt_q, domain_category, year, stat_type) is called. The existence of these radio buttons is dependent on all previous specifications, including the visualization type stored as a string viz_type and the type of statistic chosen stored as string stat_type. This means it also takes the answer to multiple data item question as a string mult_dt_q if applicable, year(s) as a list of strings year, state(s) as a list of strings state_id, commodity as a string commodity, domain as a string domain, data item as a string data_item, and possible domain categories as a list of strings domain_category (doesn't need to take in additional data items because mult_dt_q must be 'One Data Item' for the buttons to display).

The ask_barplot_xax function first checks whether the user specified bar plot or line graph, calling the function encode_viz_type(user_click), where user_click is the string representing the user's choice of bar plot or line graph. It returns True if the string is 'Line Graph' and False if the string is 'Bar Plot'. After calling this function in ask_barplot_xax, the radio buttons do not appear if True is returned. To account for all cases where domain=TOTAL and where it doesn't equal TOTAL, the function checks the validity of either item(s) in domain_category in a similar process described above, and whether 'One Data Item' was specified by mult_dt_q. It then determines whether years have been selected by the user, and if they are valid by querying the irrigation database with the get_years function in the Irr_DB class. If there aren't any valid domain categories or years, or 'One Data Item' has not been chosen, the buttons about choosing states or years for bar plots don't display. The function returns a list of strings to denote labels of the radio buttons, and two dictionaries detailing the styling where the style is set to inherit if the buttons are displayed, or where the style is set to none if they are not displayed).

For line graphs, a pair of additional radio buttons and a label for them may be displayed after a user chooses a statistic. They ask the user whether they want multiple states to be represented with multiple lines or one line if they chose an isolated data item/domain category (depending on if domain = TOTAL). To determine whether these radio buttons are triggered, and are added to the list of children built in the layout() function, the ask_linegraph_line_n(viz_type, state_id, commodity,domain, data_item, mult_dt_q, domain_category, year,stat_type) function is called. It checks whether any of the essential fields at this step (viz_type, stat_type, state_id, commodity, domain, data item) are non-iterable (not None, empty string, or empty list). It then checks for the special conditions to trigger this question, those being multiple states specified,

and either mult_dt_q = ‘One Data Item’ or only one valid (process in finding validity previously described) domain category chosen. It then determines whether years have been selected by the user, and if they are valid by querying the irrigation database with get_years. If there aren't any valid domain categories or years, or 'One Data Item' has not been chosen for domain =TOTAL, the buttons and corresponding label don't display. This is done by setting their style to none through a dictionary (key and value are both strings).

Once the necessary question is answered for the special case specific to line graphs or bar plots, three buttons (Generate Graph, Save Figure, Generate Data Table) will appear for the user. If the user’s specifications didn't trigger the special cases described above, these buttons will appear after they make their selection of statistic to display. To determine the existence of these buttons, and whether they are added to the children in the list built by layout(), display_g_or_dt_buttons(viz_type, state_id, commodity, domain, data_item, add_data_item, mult_dt_q, domain_category, year, stat_type, barax, line_n) is called.. It checks whether all required specifications for the step have been selected by the user based on their selections, including the instances in which the user has to answer targeted questions (such as multiple lines or one line for line graphs, or states or years on the x axis for bar plots). Because of this, the function takes in all of these parameters: visualization type described by a string viz_type, the statistic type described by a string stat_type, the possible answer to the x axis question for bar plots (a string barax),the possible answer to the number of lines question for line graphs (a string line_n), the state(s) selected as a list of strings state_id, year(s) as a list of strings year, commodity as a string commodity, domain as a string domain, data item as a string data_item, possible additional data items as a list of strings add_data_item, and possible domain categories as a list of strings domain_category. The function checks the validity of domain categories and additional data items entries if they are specified by the user (depending on if domain=TOTAL) since callback remembers past selections for different combinations of data specifications. Then the function checks validity of years chosen by the user according to those valid domain categories or additional data items.If there are no valid items for these fields, depending on whether domain=TOTAL or not, the three buttons do not display. Then the display_g_or_dt_buttons function checks the cases in which barax or line_n need to have an answer. If they do not in these cases, the three buttons do not display. If there are valid years given by specifications set by the user, a dictionary (key and value are strings) is returned describing the display of the final three buttons held in a html.Div (style is inherit if they are shown, style is none if they are not).

The first button whose behavior is defined in the code is the “Generate Graph” button. When clicked, it will display the user’s desired graph. To determine whether the graph can be displayed, and whether the generate graph button is disabled, the display_graph(n_clicks, viz_type, state_id, ,commodity, domain, data_item, add_data_item, mult_dt_q, domain_category, year, stat_type, barax, line_n) is called.

Since it needs access to all previous user selections, it takes in all of these parameters: visualization type described by a string viz_type, the statistic type described by a string stat_type, the possible answer to the x axis question for bar plots (a string barax), the possible answer to the number of lines question for line graphs (a string line_n), the state(s) selected as a list of strings state_id, year(s) as a list of strings year, commodity as a string commodity, domain as a string domain, data item as a string data_item, possible additional data items as a list of strings add_data_item, and possible domain categories as a list of strings domain_category.

If the generate graph button has not been clicked (indicated by an integer n_clicks) for a particular combination of user selections, the graph does not display. If the generate graph button has been clicked for a particular combination of user selections, it is disabled. The function uses ctx.triggered to determine what has been most recently changed, such as the clicking of the generate graph button.

The display_graph function uses the previous function display_g_or_dt_buttons to determine whether all required selections (accounting for all cases regarding the value for domain and the amount of states and years specified) from the user are made and valid. If not, the graph is not displayed. If all required selections are made, the function acquires applicable valid data selections to then use in a final query to the irrigation database (uses final_query and execute_final_query functions found in the Irr_DB class defined in Irr_DB.py). It constructs a unique dictionary to pass into these functions depending on the domain value, the visualization type , and the value of mult_dt_q. It also constructs unique dictionaries for the cases in which barax (states or years) or line_n (multiple lines or one line) need to exist.
It gets the result from the final query to the database and creates a line graph or bar plot depending on earlier user choice. It does so by calling make_bar_plot or make_line_graph found in visualization.py, and the graph is set to be displayed when the generate graph button is clicked. The function returns a plotly graph object, a dictionary (key and value are strings) to describe whether the graph is displayed, and another dictionary (key and value are strings) to describe whether the generate graph button is disabled (either is True or False).

When making the associated graph, the final_query(operation, params, s_multiple_or_one], yr_or_states,line_graph) is called. The parameter params is a dictionary where its keys match a column name in the irrigation database (state_id, year, commodity, domain, data_item, and possibly domain_category). Each of its values is a list of strings that pertain to the specifications set by the user in the Dash app. The parameters s_multiple_or_one and yr_or_states correlate to the special cases the user encounters after setting their desired statistic in the final Dash app, and line_graph is boolean where True indicates that the visualization type is a line graph, and bar plot if False. The function constructs a string detailing the final query to the database that utilizes json trees.

The function utilizes the SQL aggregation method (MAX, MIN, SUM, AVG) chosen by the user (it is named operation here and is the full name of the method, ex. Minimum). An additional function which_statistic(user_click) is included in the Irr_DB class and called here to encode these full names of methods to their SQL equivalents. It looks at the amount of items stored at each key in the dictionary params (keys are strings, each value is a list of strings) and the type of visualization (line_graph either True or False) and sets the GROUP BY accordingly. If necessary, the function looks at further specifications set by the user that are needed to properly set the GROUP BY. This happens if the user chose multiple states and multiple years for line graphs. The function set_line_state_groupby(params['state_id'],s_multiple_or_one), where s_multiple_or_one is either "Multiple Lines, " "One Line," or None (matches what was previously line_n in the main_dash.py file). If the user chose multiple states and multiple years, or one state and one year, the function final_query calls set_group_by_bar(params, yr_or_states) where yr_or_states is either 'States' or 'Years' or None (matches what was previously barax in the main_dash.py file), to which then the GROUP BY can be set accurately. The final_query function returns a string to be used as a query in execute_final_query(query, params, line_graph).

The set_line_state_groupby and set_group_by_bar functions determine what column the aggregation method is done over. For the set_line_state_groupby function, if it is determined that there will be multiple lines for states, it sets the GROUP BY for the sql query to state_id, year. Otherwise, it sets it to year. The set_group_by_bar function uses the value passed in with yr_or_states to return whether to set the GROUP BY for the final sql query on state_id or year. When the amount of states and amount of years are both not one or multiple, it sets the GROUP BY on the item that has the higher count of values. This function also calls an additional function check_if_time_barplot that takes the the value for yr_or_states (States or Years) and encodes it to match the column names in tMain (state_id or year).

The execute query(query, params, line_graph) function is called to obtain values for a certain sql query. It queries the database for the values to be visualized or analyzed, and is called after final_query(operation, params, s_multople_or_one, yr_or_states, line_graph). It uses the string output of final_query and passes it as a query in run_query(query, params). It also uses params, a dictionary that holds all of the user's data specifications, where each key is a string and each value is a list of strings. When line_graph is False (meaning user wants a bar plot), the function returns a list of floats. When line_graph is True (meaning user wants a line graph), returns a list of lists of floats each the length of how many years specified by the user. This is done to adhere to how traces are added using plotly.graph_objects to make a line graph.

Back to the generate graph button, it either calls make_line_graph(params, y_data, operation, s_multiple_or_one) or make_bar_plot(params, yr_or_states, y_data, operation). The function make_line_graph takes in all of the specifications set by the user in the dictionary params (each key is a string, each value is a list of strings). The keys consist of state_id,

commodity, domain, data_item, possibly domain_category, and year. The function make_line graphs also takes in s_multiple_or_one, a str 'multiple' or 'one' representing the amount of lines the user wanted to display if they chose multiple states, otherwise is None. It takes in the results of the query to the database according to the specifications in params as y_data(a list of lists of floats, where each list is the length of how many years specified by the user), and takes in the sql aggregate function (used in the query) in the form of a string passed in as operation (MIN, MAX, AVG, SUM). The function determines what is displayed in the graph according to the amount of items in the values for keys data_item and domain_category. It can also use the value stored in s_multiple_or_one if states could possibly be represented by a line in the line graph (when one data item and/or one domain category chosen). The function formats how the names of each of the lines is to be displayed in the legend by calling form_x_tick_labels(params, line_graph, data_item). If each line is a data item or domain category, the function ensures a minimal amount of horizontal space is used by using line breaks. When doing so, it also sorts them in alphabetical order to match the ordering of results from the query to the database held in y_data. If states are representing each line, the function also sorts them in alphabetical order. If multiple lines are to be displayed on the graph, the make_line_graph uses a for loop to add traces to the graph, each with its own legend group to then be able to set spacing between each of the labels for the lines in the legend. It also formats the hover text for each line added to the graph.
The make_line_graph function further determines the y axis title, finds title for overall visualization by calling get_full_title(operation, params, y_ax_title), and finds the vertical position for the title by calling set_title_pos(title). It adjusts hover text styling, title format, and spacing between items in the legend. It then returns the line graph to be placed in the final Dash app as a plotly.graph_objs._figure.Figure.

The make_bar_plot function takes in all of the specifications set by the user in the dictionary params (each key is a string, each value is a list of strings, and the keys consist of state_id, commodity, domain, data_item, possibly domain_category, and year). The make_bar_plots function further takes in yr_or_states (a str representing the column of the user's choice 'year' or 'state_id'), describing if states or years is on the x axis (otherwise is None). It takes in the results of the query of the database according to the specifications in params as y_data(a list of floats), and takes in the sql aggregate function (used in the query) in the form of a string passed in as operation (MIN, MAX, AVG, SUM). It determines the x axis based on the amount of values and existence of keys in params, or value passed in as yr_or_states, and sets the corresponding axis title. This is done by another function called name_encode_ys(yr_or_states), which converts the column name from the irrigation database (yr_or_states) to an equivalent x axis title, such as STATE or YEAR. The function ets the appropriate x tick labels and formats them to avoid text overlap by calling form_x_tick_labels(label_list, line_graph, data_item). It sets the x tick label size based on what is on the x axis, as year or states have a slightly larger tick label font size on the x axis compared to domain categories or data items on the x axis for readability purposes. When setting x_tick_labels, it sorts the list of items (held by one the keys in

params) in ascending order to match the ordering of results from execute_final_query (held in y_data). It determines what units will be on the y axis, which is stored as y_ax_title. The function also finds the appropriate title for the bar plot by calling get_full_title(operation, params, y_ax_title), and its position by calling set_title_pos(title). It then formats the hover text for all bars, with the text dependent on what is on the x axis. The function ends in creating a bar plot using plotly express, and returning the resulting bar plot to be placed in the final Dash app as a plotly.graph_objs._figure.Figure.

Both graphing functions call form_x_tick_labels(label_list, line_graph, data_item), which is also found in visualization.py. It takes the names of the tick labels as a list of strings label_list, and takes in two booleans, line_graph and data_item. It prevents overlap of text in barplots and minimization of visualizations in line graphs (too accommodating long horizontal labels). For bar plots, its adds a line break after each group of 3 words, and after the remaining couple of words if applicable. When line_graph is True, the line break for the last line is removed. The item data_item is a boolean, as it being true indicates that the labels refer to data items using the same unit. The function in this case removes all the redundant information shared across all tick labels. The overall function returns a list of strings to be used as the tick labels in a bar plot or line labels to be used in the legends of line graphs.

Both functions making line graphs and bar plots also call the get_full_title(operation, params, , y_ax_title) function. It is utilized to set titles for the visualizations, and a header for the data table with the display_table function in main_dash.py. It constructs a title based on the amount of domain categories or data items specified by the user in params (a dictionary where each key is a string and each value is a list of strings) and utilizes other values in params if necessary to construct a sufficiently specific title. In the case multiple data items were chosen by the user, it uses the y axis title passed in as y_ax_title(a string) in the overall title. To give a more accurate title for visualization/data table, it also takes in an operation (in the sql aggregate function form of either Minimum, Maximum, Average, or Sum). Items that are present in all titles are the states and years selected by the user, held in params. The function overall returns a specific title for the visualization/data table the user wants to obtain as a string.

Within get_full_title, an additional function called set_dt_title(dt_list) is called and the result is implemented in the case that there is only one data item specified by the user. The data item is held as a string in a list passed in as dt_list. Some data item names are quite long, so the function adds a line break after the 2nd comma if there are 3 or more commas. The function then Returns a string representing the properly formatted data item to be presented in the title of the final visualizations or data tables the user obtains.

The functions make_bar_plot and make_line_graph also call the function set_title_pos(title). Depending on specifications set by the user, the length of the title and how many line breaks it has changes. In order to keep the title (passed in as a string and named title)

visually appealing no matter the amount of line breaks, the function changes the vertical (y) position of the title depending on how many line breaks there are. It approximately centers the title between the top of the line graph or bar plot and the top of the entire visualization. It then returns a float t_ypos describing the vertical position of the visualization titles.

Back in the main_dash.py file, after the graphs have been generated, the user has the option to save the figure by clicking on the button named 'Save Figure'. This is done by display_save_fig_button(n_clicks, graph_clicks, output_fig). The function determines whether the Save Figure button is disabled, and saves the graph figure if the button is clicked. It takes in the number of times the save figure button has been clicked (an int n_clicks), the number of times the generate graph button has been clicked (an int graph_clicks), and the resulting figure made once the user clicks the generate graph button (a plotly graph object called output_fig). It then utilizes ctx.triggered to determine what most recently triggered the callback. If the generate graph button has not been clicked (using graph_clicks), the save figure button is disabled (using a dictionary with the key and value strings, setting the key disabled to the value True). If the most recent click was the generate graph button, the function enables the save figure button to be clicked. If the most recent click was the save figure button, the function saves the figure (output_fig) as a png to a folder called figures in a folder called user_results using the kaleido package (as described by the plotly documentation). The function names the newly created figure using the number of times the save figure button has been clicked (n_clicks), and disables the save figure button.

The last button the user can press, and then disable once clicking, for a certain set of data specifications is the generate data table button. Its disabled property is connected to the function display_table(n_clicks, viz_type, state_id, commodity, domain, data_item, add_data_item, mult_dt_q, domain_category, year, stat_type, barax, line_n) The function determines whether the data table holding the results from the query to the irrigation database is displayed or not. However, the data table must be constructed first.

The function then takes in all of these parameters: the number of times the generate data table button has been clicked n_clicks, the user specified visualization type viz_type (a string), the statistic type described by a string stat_type, the possible answer to the x axis question for bar plots (a string barax), the possible answer to the number of lines question for line graphs (a string line_n), the answer to whether user wants multiple data items in the string mult_dt_q, the state(s) selected as a list of strings state_id, year(s) as a list of strings year, commodity as a string commodity, domain as a string domain, data item as a string data_item, possible additional data items as a list of strings add_data_item, and possible domain categories as a list of strings domain_category.

The display_graph function uses the previous function display_g_or_dt_buttons to determine whether all required selections (accounting for all cases regarding the value for domain and the amount of states and years specified) from the user are made. If they are not, the graph is not displayed by using a dictionary with strings for a key and value to set the style to none. If all required selections are made, the function then acquires applicable valid selections (due to callback remembering past selections). The function further takes into account the type of visualization specified, as well as the amount of years, states, domain categories, and data items passed in to construct an appropriate dictionary to pass into the final_query and execute_query functions. In this step, it accounts for the special cases in which barax or line_n need to be defined. After final_query and execute_final_query are executed, the function defines a path for the data table to be written to, which is table_<number of times the generate data table has been clicked n_clicks>.csv.

The get_statistics(path, vals, params, yr_or_states, s_multiple_or_one ,line_graph) in data_table.py constructs a pandas DataFrame and writes it to a .csv file that is accessed within display_table in main_dash.py. In addition to the path (a string) for a dataframe to be written to, it takes in vals, either a list of floats or a list of lists of floats depending on the visualization type, a dictionary with a string as key and list of strings as value with params (corresponds with the user's selections on the Dash app), a string yr_or_states describing whether years or states are on an x axis of a bar plot, a string s_multiple_or_one describing the amount of lines depicting states for a line graph, and a boolean line_graph describing whether the visualization type is a line graph or not. From these parameters and the amount of items held in each value of the keys in params, the function determines whether to call build_stat_df_line or build_stat_df_bar on the applicable parameters, and what the first column of the resulting tables is. It defines this column name in all uppercase to match the text style in the visualizations a user can produce with the Dash app. Once getting a pandas dataframe returned, the function writes the pandas Dataframe to the path specified.

The build_stat_df_line(col_names, df_params, df_vals,param_key) function can be called by get_statistics(path, vals, params, yr_or_states, s_multiple_or_one, line_graph) when line_graph=True, indicating that the user wanted a line graph. It takes in a list object with one string as its only value called col_names that denotes the title of the first column of the data table to be made, a dictionary passed in as df_params, with each key a string and each value a list of strings (describes the specifications set by the user for state_id, commodity, domain, data item, year, and possibly domain category), a list of lists called df_vals, where each element in a list is a float (the corresponding value of a particular line on the line graph at a specific year chosen by the user) and a string called param_key to denote what key in the irrigation database each line on the corresponding line graph relates to (state_id, data_item, or domain_category). It constructs a 2-d list that is then translated into a pandas DataFrame that will be presented to the user if desired. Each list is composed of the name of a line represented on the line graph and its

corresponding values for each year specified by the user The function sorts the list of strings associated with df_params[param_key] to match the ordering of data in df_vals. After making the pandas dataframe, the function returns it to get_statistics.

The build_stat_df_bar(col_names, df_params, df_vals, param_key) can be called by get_statistics(path, vals, params, yr_or_states, s_multiple_or_one, line_graph) when line_graph=False, indicating that the user wanted a bar plot. It takes in a list with one string as the element called col_name, which represents the x axis of the bar plot the user obtains. It also takes in df_params, a dictionary where each key is a string and each value is a list of strings. Each key is a column in the irrigation database (year, state_id, domain, domain_category, data_item). This function further uses the string passed as param_key (a key name in df_params) to denote which set of strings in the dictionary represent the tick labels in the corresponding bar plot. It sorts the list of strings associated with df_params[param_key] to match the ordering of data in df_vals, a list of floats that represent the results of the query to the database according to the specifications selected by the user in the final Dash app. The function then forms a 2-dimensional list to be translated into a pandas DataFrame, where the first column title is the x axis title of the corresponding bar plot obtained by the user and the second column is Value. Each entry in the first column represents the corresponding tick labels to the bar plot obtained by the user, and each entry in the 2nd column is the corresponding y value.

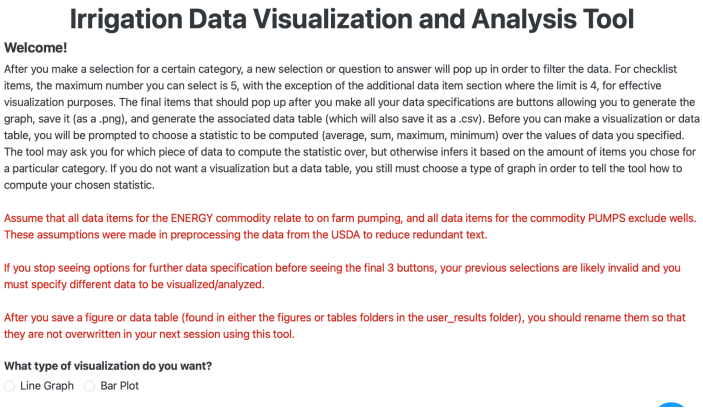
Once the table is created, an appropriate title in the form of an html label is made. Both the html.Label and data table act as the children to the item 'table-container', which is what actually is determined to be displayed or not. The html.Label (bolded) over the data table matches the title of the corresponding graph to the user's data specifications (retrieved by the get_full_title function in visualization.py). After the desired data table has been written to a .csv file, the function reads that .csv file to become a dash bootstrap table component to match the dash bootstrap theme. The table is then added to 'table-container', along with the appropriate title.

The data table and its appropriate title are created here and act as the children to the item 'table-container', which is the item to be determined to be displayed or not. If a data table is displayed, a title is also displayed over it as a html.Label matching the title of the corresponding graph to the user's data specifications (retrieved by the get_full_title function in visualization.py). To display the data table, the function writes the data table to a .csv with the get_statistics function defined in data_table.py. The name of the file is uniquely identified with how many times the generate data table button has been clicked in a session (an int passed in as n_clicks),and is saved in a folder called tables within another folder called user_results. Then reads that .csv file to become a dash bootstrap table component to match the dash bootstrap theme. The table is then added to 'table-container', along with the appropriate title, and is displayed while the button to generate the data table is disabled.

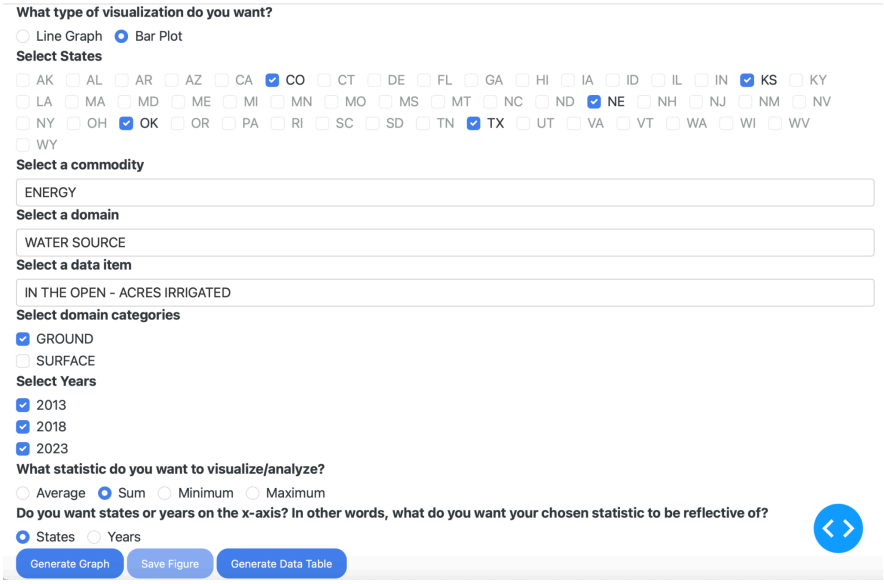
Additionally, if the generate data table button has not been clicked for a particular combination of user selections, the data table does not display. If the generate data table button has been clicked for a particular combination of user selections, it is disabled. The function uses ctx.triggered to determine what has been most recently clicked (what triggered the callback).

4 Results

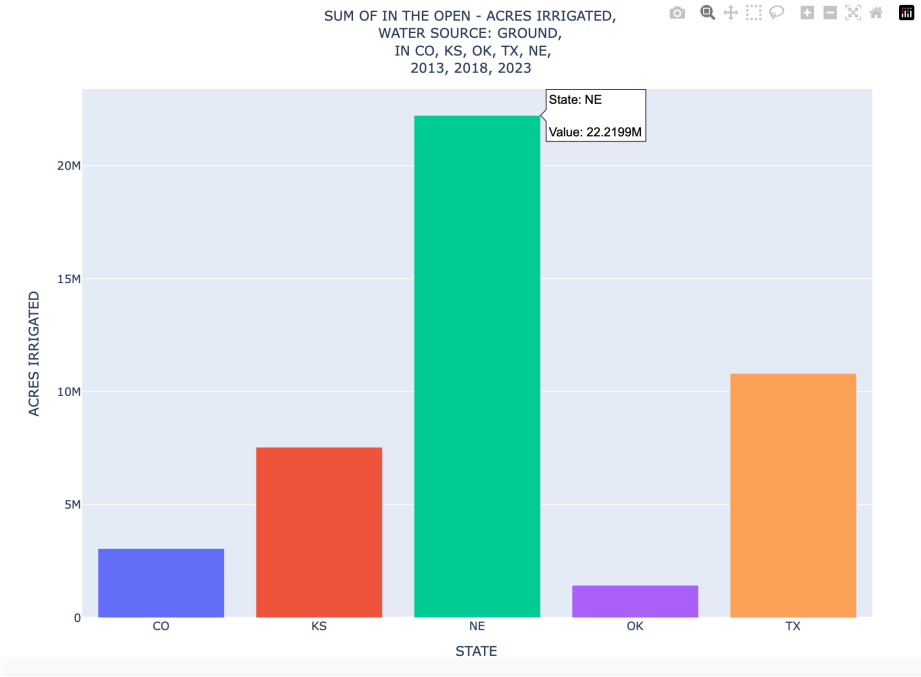
The tool's, or Dash app's, initial appearance is:



Many aspects of irrigation can be revealed by using this tool. For instance, the example of the depletion of Ogallala Aquifer discussed in the introduction of this project can be visualized. The use of ground water across 5 key states that use it, those being Colorado, Kansas, Nebraska, Oklahoma, and Texas (Scott, 2019), can be found performing the following specifications:



After clicking the “Generate Graph” button, the following was displayed, with the information box appearing only once hovering over a bar:



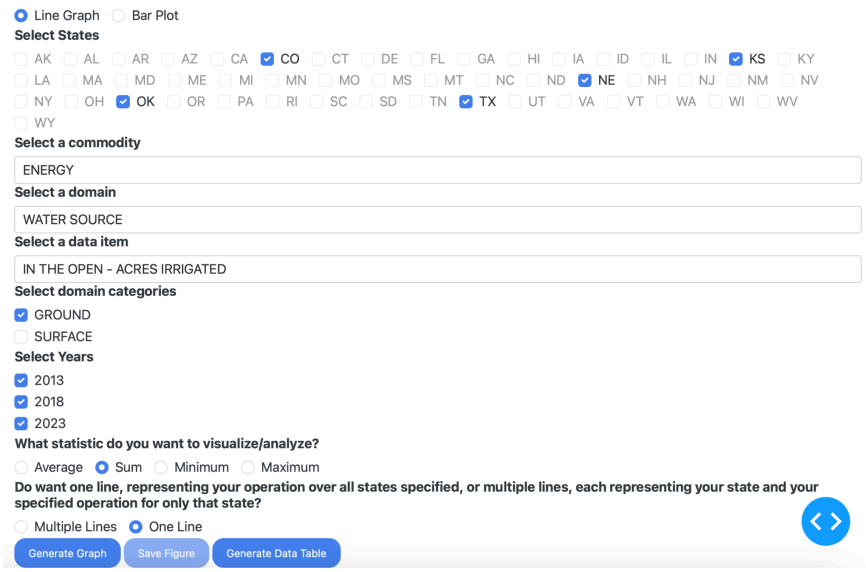
The corresponding data table was generated by clicking the “Generate Data Table” button next to the “Save Figure” button:

SUM OF IN THE OPEN - ACRES IRRIGATED, WATER SOURCE: GROUND, IN CO, KS, OK, TX, NE, 2013, 2018, 2023

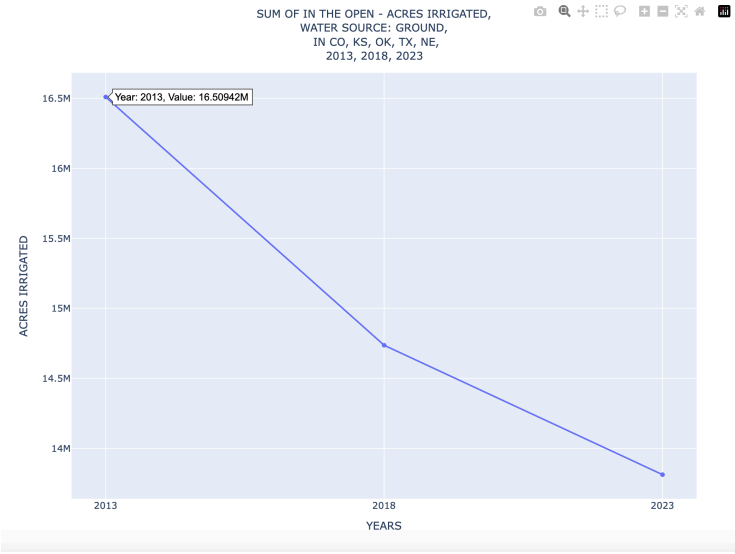
STATE	VALUE
CO	3057718
KS	7541818
NE	22219897
OK	1432628
TX	10806057

This graph and data table clearly suggest that Nebraska uses groundwater sources, such as the Ogallala aquifer, for irrigation far more than surrounding states that also use it. From this information, government officials may be inclined to enact more water conservation efforts in Nebraska to potentially reduce its groundwater use.

A line graph variant of this example, instead showing an aggregation over all the states, can be generated through these specifications:



Once clicking the generate graph button this interactive graph (hover aspect shown) appears:



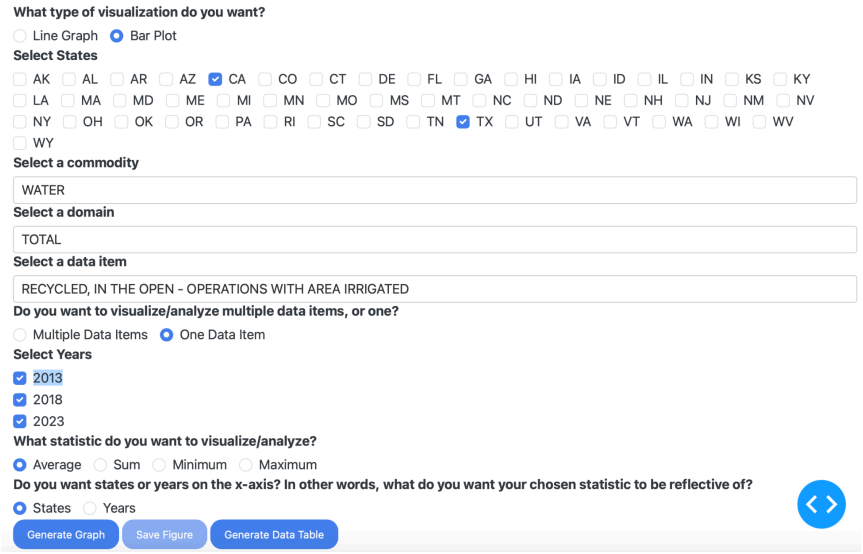
Where the associated data table is:

SUM OF IN THE OPEN - ACRES IRRIGATED, WATER SOURCE: GROUND, IN CO, KS, OK, TX, NE, 2013, 2018, 2023

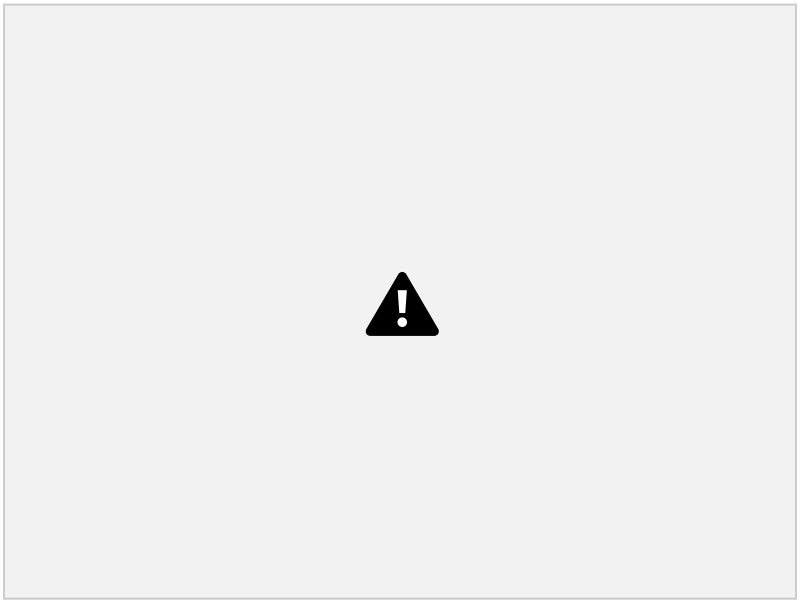
WATER SOURCE	2013	2018	2023
GROUND	16509423	14736482	13812213

From this graph and data table, it appears that the total use of groundwater has decreased over the past 10 years. This could be demonstrative of more water conservation efforts, or be indicative of further depletion of the Ogallala Aquifer.

Other interesting nuances can be found about a particular state's prioritization of sustainable irrigation practices. For instance, the following specifications can be made to compare two states with large agricultural industries, California and Texas, on the average amount of operations within each that with use recycled water in irrigation:



The generated graph (after saving the figure) and data table (presented in the Dash app) are:



AVG OF TOTAL RECYCLED, IN THE OPEN - OPERATIONS WITH AREA IRRIGATED, IN CA, TX, 2013, 2018, 2023

STATE	VALUE
CA	851.6666666666666
TX	295

Although both states have large agricultural industries, the farm owners in California much prefer to irrigate with recycled water compared to farmers in Texas. This may be indicative of the difference in public opinion about sustainable agricultural practices or possibly a difference in government incentives for irrigating with recycled water.

For these graphs and data tables, only the WATER and ENERGY commodity were filtered upon, but there are many more opportunities to unveil irrigation information about states, years, domains, data items, and domain categories for a user by using this tool.

5 Future Work

For future work on this project, geographic aspects, user accessibility, and visualization aspects can be expanded upon. For geographic aspects, the project currently uses state name data primarily. The original intent in keeping the state ANSI codes was to make it so that data analysts or government officials experienced in map making can obtain results using these numeric codes rather than by names represented by text. To employ this, there could potentially be a button at the top of the Dash app that changes the state abbreviations to the state ANSI codes and sustains that choice throughout the entire app unless the user clicks the button again. To enhance visualization aspects, there could be a choropleth option in addition to the bar plot and line graph options. A user would be forced to pick an isolated data item/domain category, but would in turn visualize the difference in values for this piece of data across the entire United States rather than a limited 5 states. Related to enhanced visualization aspects, title case for all the labels on the graphs could be utilized rather than all uppercase like it is now. To do this, some sort of function ignoring articles and prepositions in grammar, keeping them in lower case, would have to be established. For user accessibility, the hover text detailing state names in both line graphs and bar plots could be the full state names rather than the abbreviations. If these suggestions were established, the tState table in the irrigation database would be used to a greater extent than it currently is. Additionally, the user could be given more image formats to save their graphs to, as it currently only saves them to png files. Ultimately, possible future work includes enhancing the tool further with regards to how users can visualize data and understand it.

References

Hanrahan, R. (2024, January 31). *Ogallala Aquifer depletion threatening rural communities & Ag*. Farm Policy News. https://farmpolicynews.illinois.edu/2024/01/ogallala-aquifer-depletion-threatening-rural-communities-ag/

Hrozencik, A. R. (2023, September 8). *Irrigation & water use*. USDA ERS - Irrigation & Water Use. https://www.ers.usda.gov/topics/farm-practices-management/irrigation-water-use/

Little, J. B. (2009, March 1). *The Ogallala Aquifer: Saving a vital U.S. water source*. Scientific American. https://www.scientificamerican.com/article/the-ogallala-aquifer/

Scott, M. (2019, February 19). *National Climate Assessment: Great plains' Ogallala Aquifer drying out*. NOAA Climate.gov. https://www.climate.gov/news-features/featured-images/national-climate-assessment-great-plains%E2%80%99-ogallala-aquifer-drying-out#:~:text=The%20Ogallala%20Aquifer%20underlies%20parts,Dakota%2C%20Texas%2C%20and%20Wyoming

USDA. (2022). *Appendix A. census of agriculture methodology*. United States Department of Agriculture National Agricultural Statistics Service. https://www.nass.usda.gov/Publications/AgCensus/2022/Full_Report/Volume_1,_Chapter_1_US/usappxa.pdf