Irrigation Data Visualization and Analysis Tool

Kirin Mackey

Updated as of October 29, 2025

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

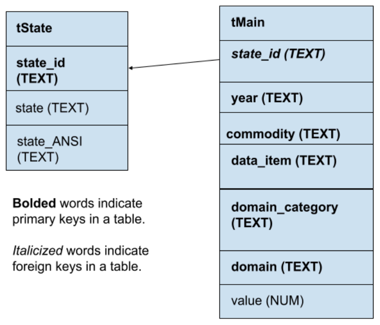
State ANSI	American National Standards Institute (ANSI) standard 2-digit state codes.
Commodity	The primary subject of interest related to irrigation (ENERGY, FACILITIES & EQUIPMENT, LABOR, PRACTICES, WATER, WELLS, PUMPS)
Data Item	Describes an aspect of the commodity measured Describes an action taken 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

3.1 Navigating the Tool

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.

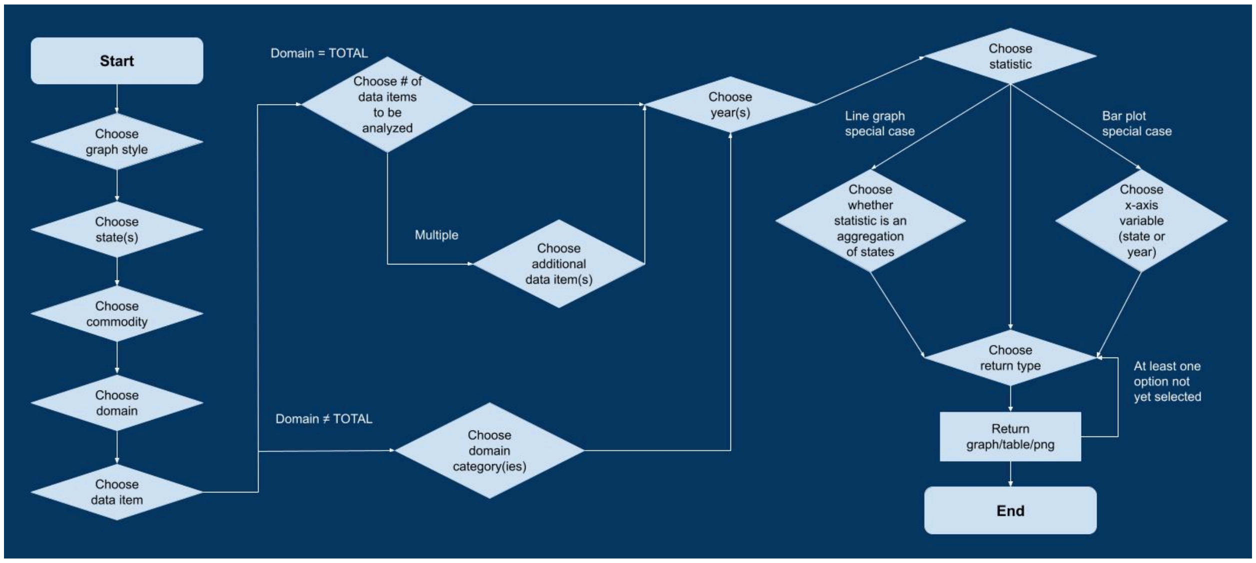
In general, the tool determines what field is being analyzed by looking at the amount of items chosen for the year, state, data item, and domain category fields. For example, if multiple domain categories have been chosen with multiple years and multiple states, the bar plot or line graph will compare 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 and has specified only one data item (domain is set to TOTAL) and possibly one domain category (domain isn't set to TOTAL). They must then either have chosen 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, 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 each 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.

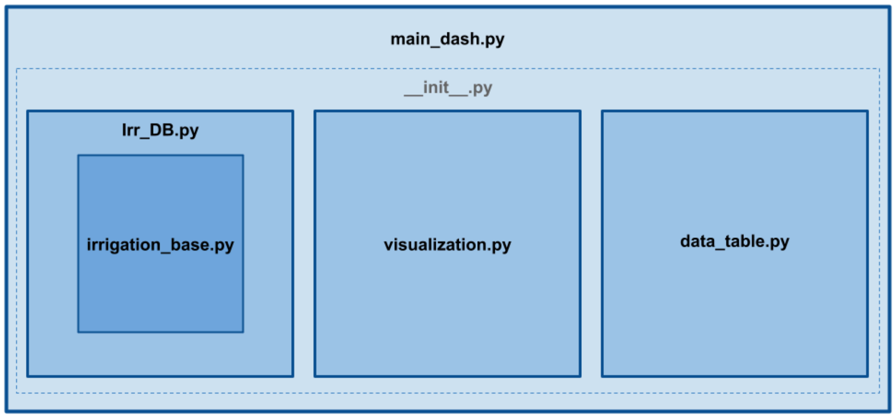
After these special case questions, if necessary, are asked, 3 buttons are displayed to the user. They allow the user to obtain a graph of their data specifications (driven by plotly), save the figure as a .png, and obtain a data table representation of their data specifications (saved as a .csv file). The save figure button is disabled until the generate graph button is disabled, and all three buttons are disabled once the user clicks them. If an earlier selection is changed by the user, the outputs of the following selections, including the existence of the buttons, are 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 different data.

Below is a visual representation of the order of choices the user must make when navigating this tool:



3.2 How the Code Works

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. A visual representation is presented below to depict the relationships between each of these files.



The __init__.py file is empty. It is created so that the other files in the src directory (Irr_DB.py, irrigation_base.py, data_table.py, visualization.py) can be treated as modules within a package, that being src. This makes it so that across all the .py files, classes and/or functions can be imported by calling src.<insert_file_name_holding_class_or_function>. Without 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, such as text items, dropdown menus, and radio buttons, by taking in a list of children. Each child represents one of the components. The layout function defines all possible components of the webpage, creates the list of children, and orders the list according to when the children are added to the webpage (first to last). 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 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 text items always appear because 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 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, where for each item there is a dictionary holding the label of the item and its associated value. For the states checklist, both the labels and values are state abbreviations, which are held in 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 item named create passed in as True. The constructor of the DB class uses the os module to check whether the path exists or not on the user's device. If it does not and create = True, it sets up a connection for the database and closes it. It also creates an attribute named self.exists and sets it to False. This is so that in the constructor for the Irr_DB class, it 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 is set to True. In this case, the build_tables() and load_tables() functions are not called, and no duplicate databases are created as a result.

When the irrigation database is being created in irrigation_base.py by the build_tables() function, the existence of tables tMain and tState is checked. If they do, they are dropped using cursors. Otherwise, empty tables for them are created through CREATE TABLE sql queries and cursors executing 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 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 type of commodity (ENERGY, FACILITIES & EQUIPMENT, LABOR, PRACTICES, WATER, WELLS, PUMPS) 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 a new entry for that row.

Next, the prep_data() function takes a subset of state data from the cleaned irrigation data, State and State ANSI. It uses the drop_duplicates() function 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 performed 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. Another dataframe called tMain is created. It holds the unique combinations of entries in the State, Year, Commodity, Data Item, Domain, Domain Category, and Value of the newly cleaned irrigation data. The columns in the dataframe named tMain match the columns specified for tMain in the ERD found in the Data Description section of this project. These new dataframes tState and tMain act as values to keys 'tState' and 'tMain' respectively and are returned to the load_data() function found in irrigation_base.py.

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(). Within load_data(), the load_table() function takes in a sql query stored as string named sql, uses the pandas function to_dict 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 main_dash.py, the get_states() function from the Irr_DB class (Irr_DB.get_states()) is called to create options for the state checklist. It queries tMain for distinct states using the function run_query(sql, params) in the DB class (defined in Irr_DB.py), where sql is the string detailing the sql query and params is any parameters needed to be defined by user. The item params in the form of a dictionary with each key a string and each value is a list of strings. The database irrigation.db 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 the function values.flatten.to_list() to get the results as a one dimensional list of strings. When creating the states checklist in main_dash.py, this list is iterated through to create dictionaries, each of which represents each item in the checklist.

The amount of states 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 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 checklist is not displayed. If the visualization type has been chosen, the checklist item is added to the webpage. Then update_multi_options(value, viz_type) limits the amount of states the user can choose by resetting the options component of 'state-cl' (the id name for the state checklist). Through dictionary comprehension, the function adjusts the disabled property of every item not selected if 5 states have already been clicked. update_multi_options(value, viz_type) returns the options for 'state-cl' as a list of dictionaries (both as keys and values are strings). For whether or not the state checklist is displayed, the function returns a dictionary object named style twice (once for the header of the section and the other for the actual checklist). The value assigned to the key ‘style’ is either 'inherit' (the state section displays) 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 layout() function. In layout(), the initial options available to the user are represented by an empty list, but the function display_coms(state_id) in the callback section changes this so that they are presented to the user. The function display_coms(state_id) takes in the user's choice of state(s) as a list of strings named 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 (two duplicate dictionaries returned have the ‘style’ key assigned the value ‘none’). If the user has done a selection of states, meaning the list state_id has a length, the commodity dropdown and header are displayed to the user (the value for the ‘style’ key in the two duplicate dictionaries returned is set to ‘block’). The options presented to the user are the results from calling Irr_DB().get_commodity(), meaning the function get_commodity() is in the Irr_DB class defined in the file 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 in irrigation_base.py are not called. get_commodity() 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(state_id) function in main_dash.py, to which then each element is an option in the dropdown the user uses to choose a commodity. 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. These items are added to the children list in the function layout() found in main_dash.py so that they are displayed to the user. 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. update_doms(state_id,commodity) takes in the values for state(s) chosen by the user as state_id (a list of strings), and the commodity chosen via the commodity dropdown (stored as a string called commodity). This is because the possible domains the user can choose from are dependent on their previous selections. The function queries the irrigation database using the Irr_DB().get_domains(comm_params) function found in the Irr_DB class.

The Irr_DB.get_domains(comm_params) function is within the Irr_DB class, which is defined in Irr_DB.py. The function takes in a dictionary comm_params, in which its keys are strings (‘state_id’ and ‘commodity’) and their corresponding values are a list of strings (only includes state_id and commodity at this step). This is done to retrieve a list of valid domains. The function creates a string object that utilizes json and json trees to query the irrigation database with the run_query(sql, params) function found in irrigation_base.py. 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 returns valid domains where the state field is the same as any item in the list of states chosen, and the commodity field is the same as the commodity specified by the user. Irr_DB.get_domains(comm_params) returns a list of strings, with each element a valid domain, to the update_doms(state_id, commodity) function in main_dash.py. update_doms(state_id, commodity) checks whether results of the query to the irrigation database have any length (meaning at least one result). If there are valid items in the results (called vals in this function), the dropdown to choose a domain is presented to the user along with its label (style is set to 'block 'in the duplicate dictionaries returned). If no results were returned, it means that states and/or the commodity field have not been selected 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. It is presented by a dropdown and label that are added to the children 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 the choices available for data item are dependent on the state(s), commodity, and domain specified by the user. The function 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 only 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. get_data_items(dt_params) 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 the function update_dts(state_id, commodity, domain) 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 'block' is assigned its style.

After the data item has been selected, the branching off point with regards to whether domain =TOTAL is created. 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 (id being 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 use the same units as the initial data item selected by the user. The user is 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), which 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. This makes it so state selection is a list of strings state_id, commodity is a string named commodity, domain is a string named 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.

The function update_mult_dts_items 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.

To find the data items that can be compared with the initial data item chosen, not including those held in init_vals, 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. After checking whether the domain field was set to total, it returns an encoded version of the user's previous choice to 'multiple' or 'one'. Back in get_domain_categories(dc_params, mult_dt_q), if the result of number_dt_question is ‘one’ the function returns nothing (the user chose ‘One Data Item’). If ‘multiple’ is returned by the encoder, the intermediate_domain_categories(idc_params) function, also within the Irr_DB class in Irr_DB.py, is called to query the irrigation database.

The intermediate_domain_categories(idc_params) function takes in the dictionary named dc_params passed into get_domain_categories (naming it now 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. It does so by adding another key to the dictionary idc_params 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 because ‘none’ is set as its style when being added to the webpage. If there are results, they are returned to the get_domain_categories function, and then the update_mult_dts_items function in main_dash.py as a list of strings.

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. The items in the checklist are determined by the function update_dc(state_id, commodity, domain, data_item, init_vals) in the callback section of main_dash.py. The arguments state_id, commodity, and data_item all refer to selections in the previous sections of the webpage. These items are placed into a dictionary, where each value associated with a key 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. The parameter mul_dt_q is set to None because the question for displaying multiple data items, one for only when the user selects TOTAL as the domain, was not triggered.

get_domain_categories(dc_params, mult_dt_q) sets the appropriate sql query utilizing json trees so that lists with multiple items can be parameters. The results of this query executed by run_query(sql, params) are returned as a list of strings to update_dc(state_id, commodity, domain, data_item, init_vals). However, callback remembers past selections of domain categories for different previous specifications, making it so invalid domain categories could be presented to the user for their current selections. To stop this, the function determines the already valid domain categories chosen by comparing two lists. These lists are init_vals, a list of items remembered through callback, and the valid domain categories from the sql query set by get_domain_categories(dc_params, mult_dt_q). The function then finds the valid domain categories already chosen for the current data specifications set by the user. Once the amount of valid domain categories already chosen by the user reaches 5, the rest of the items in the checklist are disabled by setting their disabled property to True through dictionary comprehension.

After at least one item is chosen in the checklist relating to either additional data items (domain = TOTAL) or domain categories (domain ≠ TOTAL), the user will be presented with another checklist to select the years they want their final visualization/data table to reflect. This checklist is added to the list of children created in layout(), but its options and appearance are 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 from. The valid years are dependent on all previous selections, so it accounts for cases in which domain = TOTAL (includes choice of multiple data items or one data item), and when domain ≠ TOTAL. No matter the value for the parameter domain, if any essential specifications (state(s), commodity, domain, data item) haven't been chosen by the user, the style of the checklist is set to None and it doesn't display.

For domain = TOTAL, mult_dt_q must have a value of either ‘One Data Item’ or ‘Multiple Data Items’ as they are the options presented to the user when they choose this domain path. If there is no value assigned to mult_dt_q in this case, the year checklist is not displayed. For ‘Multiple Data Items’ the function checks whether valid additional data items, a list of strings passed in as add_data_item, have been chosen by calling get_domain_categories and inputting the values for state_id, commodity, domain, and data_item also passed in. If no valid additional data items have been chosen, the year checklist is not displayed. This process of checking valid additional data items is not done for the user choice of ‘One Data Item’ since the single value passed in as data_item is already valid.

For domain ≠ TOTAL, the function checks whether the strings stored in the list data_item are valid for the current data specifications. This combats the issue of callbacks saving past data_item selections for past user specifications that yield different results for the data item checklist. Much like the case of domain = TOTAL and multiple data items, get_domain_categories is called, where the values for state_id, commodity, domain, and data_item are passed in. The list of domain categories already chosen by the user, held in domain_category, is compared to the list of possible valid domain categories for the current data specifications (returned by get_domain_categories). If no valid domain categories have been chosen by the user, the year checklist is not displayed.

Years chosen for different data specifications are stored in init_vals in order to address the previously mentioned issue of callbacks saving past user choices. After performing the checking procedures for the items required for when domain = TOTAL or when domain ≠ TOTAL, a dictionary is constructed that holds all the required parameters to query the irrigation database for valid years. 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 method of using json trees to hold lists, which then act as parameters for sql queries, is used once again. Because of json tree structure and the IN() sql operator that essentially acts as the OR operator in this case, it was found that run_query(sql, params), where params=year_params, would return invalid results. The years returned would be valid for one state, even if it wasn't valid for another if the user chose multiple states. The same phenomenon occurred for keys (parameters for the sql query) that have values that are lists holding more than one element (possibly data item and domain category depending on user domain selection).

To address this issue, the function each_choice_year(key_name, year_params), located in Irr_DB.py, is called. For this function, key_name either refers to state_id, data_item in the case domain = TOTAL and multiple data items had been chosen, or domain_category in the case domain ≠ TOTAL. These are all the possible parameters where the user can have multiple options selected. For each entry in a particular key's associated 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 the entries in particular key's associated list. It does so by utilizing numpy.unique() and counting the number of times each unique value appears in a list that contains every result of valid years for each element in the particular key's associated list (state, data item, or domain category). If the amount of times a unique year value occurs in this compilation of years list matches the amount of total items in the particular key's associated list, the year is deemed valid for the particular data category passed in as key_name. The function then returns this list of valid years for a particular data category to the get_years function.

Once back in the get_years 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. To do this, the function uses set intersection to find the common years between all lists (valid for states chosen, valid for data items if applicable, and valid for domain categories if applicable).The function sorts the final list with common years amongst the described data categories in ascending order. This is because once becoming a set during set intersection, the order of list elements does not matter. This is done to match up the labels of years presented to the user to the final data results (visualization or table) that are displayed, as SQL presents the results to be visualized/tabularized by default in ascending order. To account for the possibility of more than 5 years being deemed valid, the process described for disabling certain checklist items once 5 are selected occur here as well (using the init_vals argument once again). 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. Because of this, the function checks whether the essential specifications (state(s), commodity, domain, data item, year) have been made, and whether the specifications needed for the various cases when domain = TOTAL, and when domain ≠ TOTAL have been made as well. The process for checking the validity of possible additional data categories, mentioned in the update_years function description above, occurs. The validity of years chosen by the user is also determined, as the callback remembers year selections for different data specifications as well. If all required conditions have been met and valid years have been selected by the user, radio buttons with the options Sum, Minimum, Maximum, Average, are presented to the user (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 described in section 3.1 of this report 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 states or years on the x-axis. To determine whether these radio buttons are presented to the user when 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) in main_dash.py 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. Moreover, in the case domain ≠ TOTAL, the user must choose '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 domain ≠ TOTAL, the function checks the validity of the additional choices the user must make in these paths (process previously mentioned in the description for the function update_years). It additionally checks whether 'One Data Item' was specified if the user chose domain=TOTAL. 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 years or items particular to the domain choice, or 'One Data Item' has not been chosen, the buttons presenting the choice of states or years for the x-axis in bar plots don't display. The function returns a list of strings to denote labels of the radio buttons (empty if buttons are not displayed), and two dictionaries detailing the styling (style is set to ‘inherit' if the buttons are displayed, or ‘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 (statistic chosen is employed for each state), or one line (statistic chosen is employed over all states) if they chose only 1 data item/domain category (depending on if domain = TOTAL or not). To determine whether these radio buttons are displayed when 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.

ask_linegraph_line_n checks whether any of the essential fields at this step (viz_type, stat_type, state_id, commodity, domain, data item) are non-iterable and therefore exist (meaning 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 ‘One Data Item’ chosen when domain =TOTAL or only one valid domain category chosen when domain ≠ TOTAL (process in finding validity mentioned in the update_years function description). 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. If any of these conditions (existence/validity of items) prove to be false the buttons and corresponding label don't display by setting ‘none’ as their style.

Once the necessary question for the special case in either bar plots or line graphs is answered, 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 presence of these buttons when 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 should be noted that these three buttons are held in an html.Div object, and that object is what is added to the list of children built in layout().

The function display_g_or_dt_buttons checks whether all previous required specifications for the step have been selected by the user, including the instances in which the user has to answer targeted questions (multiple lines or one line for line graphs, or states or years on the x-axis for bar plots). The function further checks the validity of domain categories or additional data item entries (depending on if domain = TOTAL) due to the callback issue (process in the update_years function description). Then the function checks the validity of years chosen by the user according to those valid domain categories or additional data items. Then the display_g_or_dt_buttons function checks the cases in which barax or line_n (answers for the special case question for either bar plots or line graphs) apply and therefore need to have a value associated with them. If there are no valid items for these fields dependent on domain choice, the three buttons do not display.If they do not in these cases, the three buttons do not display. If the existence or validity of any of the items described above are false, the three buttons do not display by setting 'none' as the style of html.Div object holding all of them. Otherwise, the style is set as 'inherit' and the buttons display. The behavior of each of the buttons is described below.

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 then 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) function is called.

If the "Generate Graph" button has not been clicked at all (indicated when n_clicks = 0) 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 item on the webpage has been most recently changed/clicked. If the generate graph button has been clicked most recently, the graph will display and the "Generate Graph" button will disable. If a different item is what has been clicked most recently, it will not. The "Generate Graph" button may still be available to click if the display_g_or_dt_buttons function allows the html.Div object holding all three final buttons to appear (if the style for it is set to 'inherit').

The display_graph function uses the previous display_g_or_dt_buttons function 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 the required selections have been made, the display_graph function further validates the user's choices to accommodate the callback issue and creates a dictionary holding information for the final query to the irrigation database in order to create a graph. For retrieving the final information needed to create the graph, a sql query must be constructed. To do so, display_graph calls final_query(operation, params, s_multiple_or_one, yr_or_states,line_graph) found in Irrigation.py.

In final_query(operation, params, s_multiple_or_one, yr_or_states,line_graph), operation indicates the statistic the user chose, params is the dictionary of parameters used to query the irrigation database, s_multiple_or_one and yr_or_states correlate to the special cases the user encounters after setting their desired statistic, and line_graph is a boolean where True indicates that the visualization type is a line graph, and bar plot if False. The parameter operation refers to SQL aggregation methods (MIN, MAX, AVG, SUM) that the user chooses when prompted to choose a statistic. However, since the options presented to the user are the full name of each method (Minimum vs. MIN), an additional function named which_statistic(user_click) is called to convert the user statistic choice (passed in as user_click) to its SQL equivalent.

After receiving the SQL aggregation method, final_query builds the appropriate SQL query that encapsulates the user's current valid choices. It looks at the visualization type and the amount of items stored at each key in the dictionary params and sets the GROUP BY statement accordingly. The GROUP BY statement indicates which column of the database the aggregation method is to be executed over. If necessary, the function looks at further specifications that are needed to properly set the GROUP BY. A case in which this happens is if the user chose multiple states and multiple years for line graphs. To address this, final_query calls the function set_line_state_groupby(params, s_multiple_or_one), where s_multiple_or_one is either "Multiple Lines" or "One Line" in the case multiple states were chosen, or None in the case only one state was chosen. The GROUP BY statement is set to include states and years for "Multiple Lines," or to only year if "One Line" was chosen in the case of multiple states, or only one state was specified by the user.

Another case in which final_query needs to look at further specifications to accurately build the GROUP BY statement is if the user chose a bar plot visualization type, along with multiple states and multiple years, or one state and one year. Along with these conditions, the user must have only chosen 1 domain category if domain ≠ TOTAL, or only 1 data item if domain = TOTAL. Earlier when this case was triggered, the user had to choose which parameter was on the x-axis (state or year). To reflect this in a SQL query GROUP BY statement, the function final_query calls set_group_by_bar(params, yr_or_states). yr_or_states is either 'States' or 'Years' in the special cases described above, or None which indicates the app does not need further user input when deciding what parameter is represented on the x-axis. Based on the user's answers, the GROUP BY is set accordingly.

After the GROUP BY statements have been added to the SQL query built in final_query, the function returns a string to be used as a query in the function execute_final_query(query, params, line_graph). The function run_query is then performed using the SQL query built and the parameters passed in, returning the results as a pandas dataframe. The parameter line_graph indicates the visualization type chosen by the user, where if True the function properly groups the data associated with the years chosen by the user, and matches it to the type of data that is indicated by each line on the final line graph. If line_graph is False, the function only looks at the final column of the pandas dataframe returned which holds the results to be visualized. The additional step regarding line graphs is done to adhere to how lines are added to plotly line graphs.

Execute_final_query, defined in Irr_DB.py, returns the results to be visualized/tabularized to display_graph, found in main_dash.py. Depending on whether the user chose a line graph or bar plot, the function calls make_line_graph(params, y_data, operation, s_multiple_or_one) or make_bar_plot(params, yr_or_states, y_data, operation), both of which are defined in vizualization.py.

The function make_line_graph takes in all of the data specifications set by the user in the dictionary params (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 which indicates how many lines the user wants to display, and y_data which holds the results of the query to the database according to the specifications in params as (a list of lists of floats, where each list is the length of how many years specified by the user). It also takes in operation, the SQL aggregate function (MIN, MAX, AVG, SUM) specified by the user in order to form the graph title. make_line_graph determines what is displayed in the graph according to the amount of items in the values matched to the keys named data_item and domain_category from the dictionary params, as well as the value held in s_multiple_or_one if states could possibly be represented by a line (when only one data item and/or one domain category chosen). Once determining this, the function adds lines to the plotly graph object initially made when this function is called. 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 allow spacing modification between each of the line labels in the legend. It also formats the hover text for each line added to the graph.

In addition to adding lines to the plotly graph object, make_line_graph calls form_x_tick_labels(label_list, line_graph, data_item), also in visualization.py. form_x_tick_labels accounts for both line graphs (line_graph = True) and bar plots (line_graph = False). Here, data_item is a boolean indicating whether each line or bar in the visualization chosen by the user is a data item (True) or not (False). This is done to combat issues of redundancy in labeling units. This issue is only present when each data category (line or bar) in the final visualization is a data item (domain = TOTAL), as data items in the irrigation database have a different information format compared to domain categories. Additionally, label_list is a list of strings holding the names of the tick (bar plot) or line (line graph) labels, which is determined in either make_line_graph or make_bar_plot depending on the visualization type chosen by the user. Both of these functions sort the list of data categories to be used as tick (bar plot) or line (line graph) labels in alphabetical order to match the order of results returned by execute_final_query (held in y_data) when they call form_x_tick_labels.

In the case of line graphs, form_x_tick_labels prevents the minimization of visualizations, as plotly by default creates long horizontal labels to accommodate long names of data categories. In the case of bar plots, form_x_tick_labels prevents overlapping text due to a similar default tick label configuration in plotly. For each x tick label in label_list, form_x_tick_labels 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 for aesthetically pleasing spacing in the line graph's legend. The function then returns the modified list of labels with appropriate spacing (used as tick labels in a bar plot, or line labels in the legend of a line graph) to make_line_graph or make_bar_plot.

After adding the lines and their corresponding labels to a legend, the make_line_graph function further determines the y-axis title, the title of the overall visualization, and the vertical position for the title. To retrieve the title for the y-axis, the function isolates the value associated with the user's choice of data item and stores the unit listed within it. To obtain the title for the overall visualization, make_line_graph calls get_full_title(operation, params, y_ax_title), where operation is the SQL aggregation method chosen by the user, params is the dictionary holding the various user data specifications, and y_ax_title is the isolated units to be listed on the y-axis. The title returned by get_full_title is then passed into set_title_pos(title) in order to set an appropriate vertical position for the amount of line breaks the title has. Both make_line_graph and make_bar_plot call get_full_title and set_title_pos when finding the titles and their vertical positioning for the corresponding visualizations they return to main_dash.py.

The titles generated by this get_full_title attempt to be as detailed as possible in order to accurately depict the user's data specifications. However, the data item entry, if the user had chosen only one (domain ≠ TOTAL, or domain = TOTAL and only one data item chosen), can be quite long and therefore need line breaks when being added to the final title. To add these line breaks, the function calls set_dt_list(dt_list), where dt_list is the list with a single string detailing the data item choice. The function splits the data item string according to the amount of commas in it, and adds a line break after the second comma if there proves to be three or more. It then joins the string segments together and returns the modified string back to get_full_title. The final title is then created utilizing all the relevant information chosen by the user (statistic, modified data item, domain, domain category if only 1 chosen, states, years). In the case the user chose multiple data items, and therefore the result of set_dt_list is inaccurate (only accommodates 1 data item), the final title will include the units shared amongst the various data items, along with the chosen statistic, commodity, states, and years. It will further indicate each value (line or bar) represents a total.

The vertical positioning of the titles in either the generated line graphs or bar plots is determined when either make_line_graph or make_bar_plot call set_title_pos(title) after retrieving title from get_full_title. Since the titles returned by get_full_title may have different amounts of line breaks, there is no universal vertical position to make the titles look visually appealing. Depending on if there are less than, more than, or exactly 3 line breaks, a different vertical position for the title is returned as a float. These different vertical positions approximately center the title between the top of the line graph or bar plot and the top of the entire visualization. The particular value deemed appropriate is then used in make_line_graph or make_bar_plot when they each update the styling of the titles they use for the respective visualizations they generate.

Back in make_line_graph, after adding lines, appropriate line labels, and retrieving an appropriate title and positioning of the title, the function adjusts hover text styling, title format, and spacing between items in the legend. It then returns the line graph to display_graph in main_dash.py, where it will then be placed in the final Dash app as a plotly.graph_objs._figure.Figure.

If the user did not choose a line graph as their desired visualization, after execute_final_query is performed and the final numerical results are found, make_bar_plot(params, yr_or_states, y_data, operation), found in visualization.py, is called by display_graph in main_dash.py.

make_bar_plot takes in all of the specifications set by the user in the dictionary params yr_or_states, describing if states or years is on the x-axis (otherwise is None), the results of the query to the irrigation database y_data, and 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 and existence of values in params, or value passed in as yr_or_states (addresses the different domain paths and special cases for barplots), and sets the corresponding axis title. For specifically the special cases for barplots (multiple states and years chosen or one state and one year chosen, with no possibility data items nor domain categories can represented by the x-axis), another function called name_encode_ys(yr_or_states), is called. This converts the column name from the irrigation database (yr_or_states is either 'state_id' or 'year') to an appropriate x-axis title (STATE or YEAR).

After finding the correct x-axis title, make_bar_plot sets the appropriate x tick labels and formats them to avoid text overlap by calling form_x_tick_labels(label_list, line_graph, data_item). This function's performance is described above, along with how it handles line graphs. Once form_x_tick_labels returns the appropriate list of labels, make_bar_plot assigns the label sizes according to the data represented on the x-axis. The categories of year or states are assigned a slightly larger font size on the x-axis compared to domain categories or data items for readability purposes. The function then determines what units will be on the y-axis by isolating the unit detailed in the user's data item choice(s). 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). Both of these functions are described above when traversing through the line graph path. Once finding the appropriate title and positioning, make_bar_plot formats the hover text for all bars, with the text dependent on the x-axis data. The function ends in creating a bar plot using plotly express, and returning the resulting bar plot to display_graph in main_dash.py, where it will then be placed in the final Dash app as a plotly.graph_objs._figure.Figure.

After the display_graph function receives a visualization from either make_line_graph or make_bar_plot, displays it, and adjusts the "Generate Graph button" to disable, the user has the option to click the "Save Figure" button to download their newly made visualization to their computer. The performance of this button is executed by display_save_fig_button(n_clicks, graph_clicks, output_fig), a function found in main_py.

display_save_fig_button takes in the number of times the "Save Figure" button has been clicked (n_clicks), the number of times the "Generate Graph" button has been clicked (graph_clicks), and the resulting figure made once the user clicks the "Generate Graph" button (a plotly graph object called output_fig). It returns a boolean describing the disabled attribute of the button item when it is added to the webpage via the function layout(). A boolean True is returned if the "Generate Graph" button has not been clicked, or if ctx.triggered determines the most recent item that has triggered the callback is the graph figure (the graph has changed via a new user data specification). True may also be returned if ctx.triggered determines the most recent item clicked is the "Save Figure" button. In this case, the function also 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). False is only returned by the function, therefore allowing the "Save Figure" button to be clicked, if the "Generate Graph" button has been clicked for a newly and successfully generated graph.

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 also determines whether the data table holding the results from the query to the irrigation database is displayed or not, along with its title. Because of this, the function, in addition to the disabled property of the "Generate Data Table" button, returns the data table and title as children to a html.Div container within the webpage. Whether the data table and its corresponding title are displayed is also returned as the style to the html.Div container holding them.

display_table first determines whether the "Generate Data Table" button has been clicked at all (n_clicks), and what item has been clicked most recently via ctx.triggered. For the first set of valid data specifications, n_clicks must be 0 in order for the button to be available to click. To accommodate cases in which the user changes their previous data specifications, thus changing the resulting data table, ctx.triggered must show the recent item clicked is the "Generate Data Table" button for the button to disable.

Next, display_table checks whether all the required data specifications (accounting for all cases regarding the value for domain and the amount of states and years specified) have been made so that a data table can be constructed. It does so by calling the previous function display_g_or_dt_buttons described above. As a result, display_table needs to have all parameters that are needed for this validity checking. If this validity checking proves that the grouping of buttons container is not displayed, the data table and its title are not either. Otherwise, display_graph then retrieves the user's valid data choices using processes described previously and determines which path the user chose with regards to their domain choice, visualization type, and the special cases associated with each visualization type. Once doing so, the function constructs a unique .csv file name for the particular session on the webpage (table_<number of times the "Generate Data Table" button has been clicked n_clicks>.csv) so that a table can be written to it and therefore be saved on the user's device. Using the valid user data choices, display_table executes both final_query and execute_final_query (detailed above) to get the final numerical results to be tabularized.

After obtaining the numerical results, display_table constructs the data table by calling the function get_statistics(path, vals, params, yr_or_states, s_multiple_or_one, line_graph), found in data_table.py. path holds the name of the file the table is to be written to, params is the dictionary holding all of the user's data selections, yr_or_states and s_multiple_or_one address the special cases for each type of visualization, and line_graph indicates the type of visualization chosen by the user. From these parameters, get_statistics determines the title of the first column in the final data table, along with what additional function needs to be called to properly format the data table, as line graphs and bar plots differ in this aspect. In the case of line graphs, the function calls build_stat_df_line(col_names, df_params, df_vals,param_key), or build_stat_df_bar(col_names, df_params, df_vals, param_key) in the case of bar plots. Both of these functions are found in data_table.py and return a pandas dataframe object.

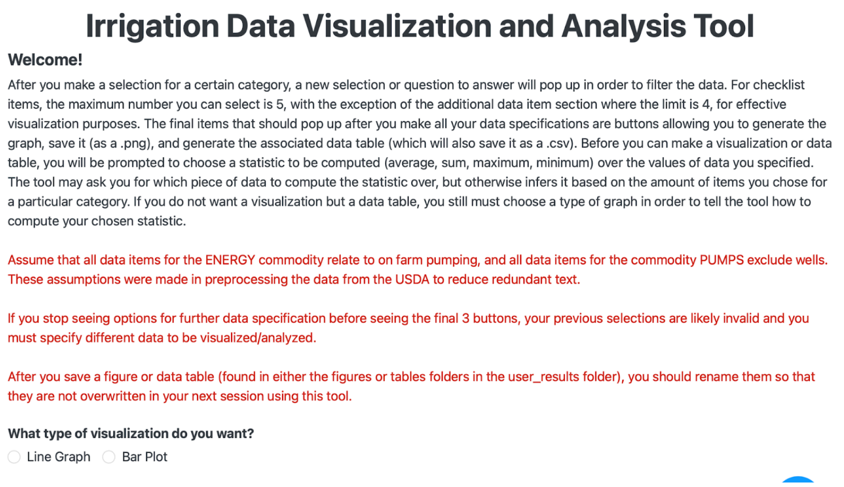
For line graphs, build_stat_df_line first uses the list with one string passed in as col_names (found earlier in get_statistics), and sets the rest of the column names according to the years selected by the user (found in the dictionary df_params holding all of the user's data choices). The df_vals argument is a list of lists, where each individual list holds the data associated with a particular year chosen. The function then constructs each row to be added to the final table, where the first value is the name of a line represented on the line graph (ex. A state name if each line represented a state's change in data over time). These names are found by using param_key, a string passed in that indexes the dictionary of user data choices found in df_params.The rest of the values in each row are the corresponding values for each year specified by the user for that particular line on the line graph. Each row of these rows are represented by a list (ex. [state name, state data for 2013, state data for 2018, state data for 2023]), and are then all added as elements to a different list. A 2-d list is then created once all rows have been determined, and build_stat_df_line converts this list of lists to a pandas dataframe and returns it to get_statistics.

For bar_plots, build_stat_df_bar similarly uses the argument col_names, a list with one string representing the name of the x-axis, to start a list of column names for the data table. The only other element added to this list describing column names is “VALUE” due to the format of bar plots. Units are not included in this because the title of the data table will detail them. Once again like build_stat_df_line, build_stat_df_bar creates a list of lists, where each list represents a row in the data table to be generated. In every list, the first element is the name corresponding to each bar (or tick label) in the bar plot. These are found using the same process utilizing param_key and df_params in build_stat_df_line. The only other element in each list constructed is the particular bar’s corresponding value found in df_vals, a list of floats passed in that holds the results of the final query to the irrigation database. The 2-d list is then converted to a pandas dataframe and returned to get_statistics.

Back in get_statistics, once receiving the pandas dataframe returned by either build_stat_df_line or build_stat_df_bar, the function writes the dataframe to the .csv path passed in as path, essentially saving it to the user’s device. Once this process is completed, the function returns None to the display_table function in main_dash.py. display_table then obtains the dataframe by reading it, and calls get_full_title found in visualization.py to accurately label the data table and match it to its associated visualization type. After obtaining the title, display_table replaces the line breaks with spaces for aesthetic purposes, and creates a html.Label object with this title. Furthermore, display_table creates a dash bootstrap table component from the dataframe read in. The label object and the table object are put into a list, which is then returned as a list of children for a container (an html.Div object) within the webpage. They are subsequently displayed for the user to hover over and analyze. The “Generate Data Table” button is also disabled, unless the user changes earlier data specifications which then results in this overall process of creating visualizations and data tables, accommodating different paths and addressing callback problems, to commence once again.

4 Results

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



Irrigation Data Visualization and Analysis Tool

Welcome!

After you make a selection for a certain category, a new selection or question to answer will pop up in order to filter the data. For checklist items, the maximum number you can select is 5, with the exception of the additional data item section where the limit is 4, for effective visualization purposes. The final items that should pop up after you make all your data specifications are buttons allowing you to generate the graph, save it (as a .png), and generate the associated data table (which will also save it as a .csv). Before you can make a visualization or data table, you will be prompted to choose a statistic to be computed (average, sum, maximum, minimum) over the values of data you specified. The tool may ask you for which piece of data to compute the statistic over, but otherwise infers it based on the amount of items you chose for a particular category. If you do not want a visualization but a data table, you still must choose a type of graph in order to tell the tool how to compute your chosen statistic.

Assume that all data items for the ENERGY commodity relate to on farm pumping, and all data items for the commodity PUMPS exclude wells. These assumptions were made in preprocessing the data from the USDA to reduce redundant text.

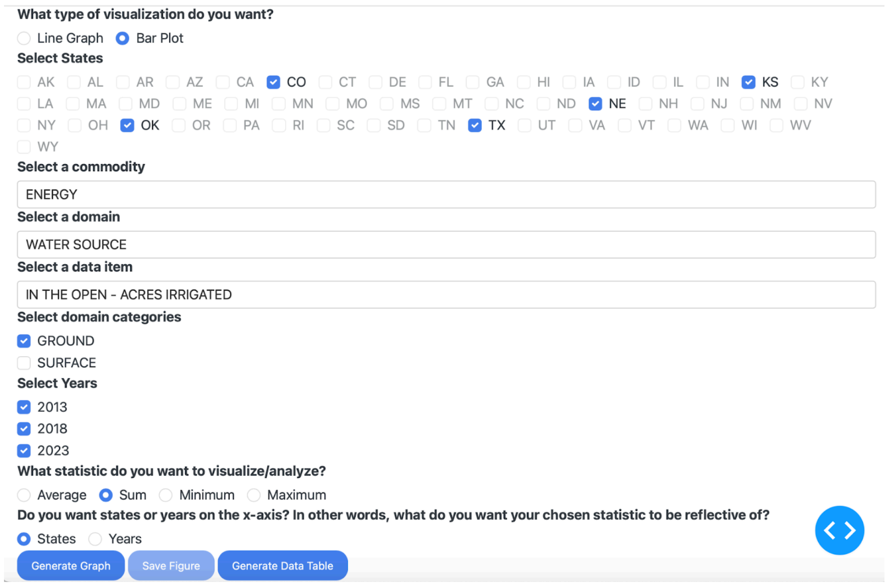
If you stop seeing options for further data specification before seeing the final 3 buttons, your previous selections are likely invalid and you must specify different data to be visualized/analyzed.

After you save a figure or data table (found in either the figures or tables folders in the user_results folder), you should rename them so that they are not overwritten in your next session using this tool.

What type of visualization do you want?

Line Graph Bar Plot

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:



What type of visualization do you want?

Line Graph [✓ Bar Plot]

Select States

AK AL AR AZ CA [✓ CO] CT DE FL GA HI IA ID IL IN [✓ KS] KY LA MA MD ME MI MN MO MS MT NC ND [✓ NE] NH NJ NM NV NY OH [✓ OK] OR PA RI SC SD TN [✓ TX] UT VA VT WA WI WV WY

Select a commodity

ENERGY

Select a domain

WATER SOURCE

Select a data item

IN THE OPEN - ACRES IRRIGATED

Select domain categories

[✓ GROUND]
SURFACE

Select Years

[✓ 2013]
[✓ 2018]
[✓ 2023]

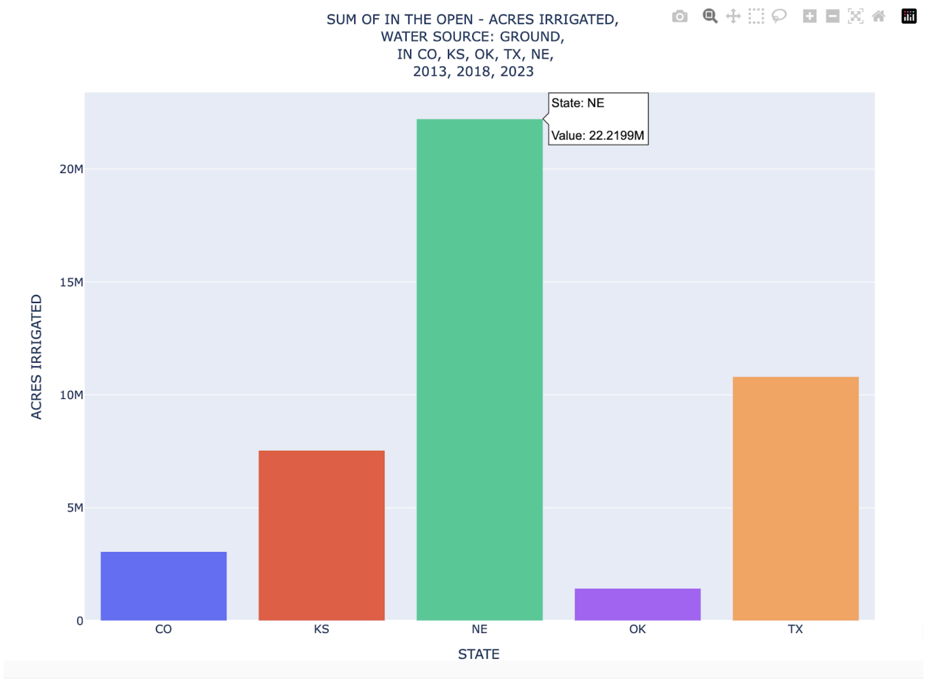
What statistic do you want to visualize/analyze?

Average [✓ Sum] Minimum Maximum

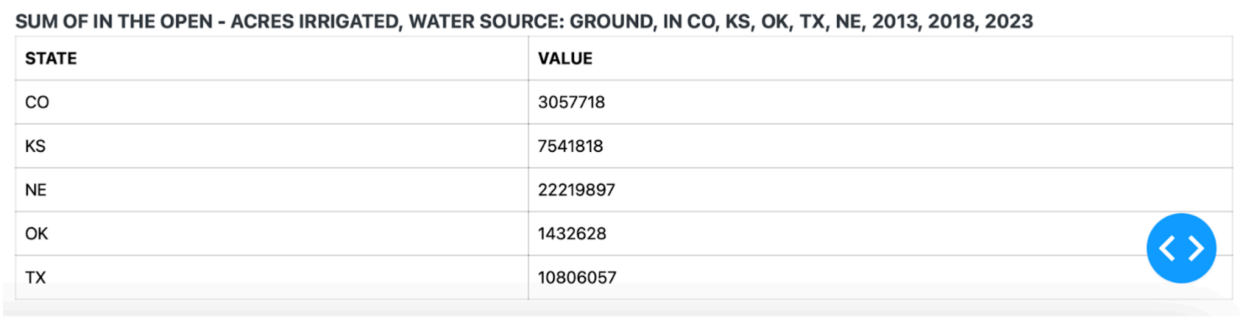
Do you want states or years on the x-axis? In other words, what do you want your chosen statistic to be reflective of?

[✓ States] Years

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



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

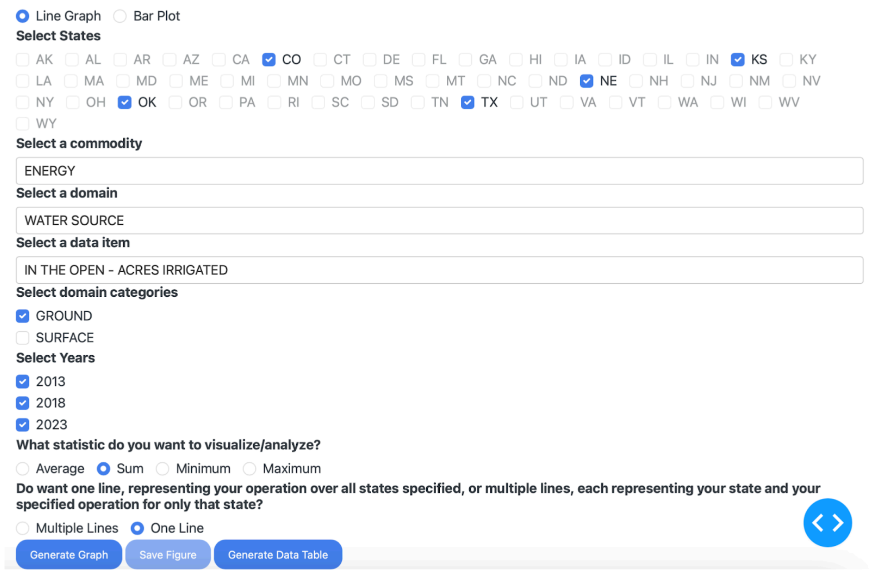


SUM OF IN THE OPEN - ACRES IRRIGATED, WATER SOURCE: GROUND, IN CO, KS, OK, TX, NE, 2013, 2018, 2023

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:



[✓ Line Graph] Bar Plot

Select States

AK AL AR AZ CA [✓ CO] CT DE FL GA HI IA ID IL IN [✓ KS] KY
LA MA MD ME MI MN MO MS MT NC ND [✓ NE] NH NJ NM NV
NY OH [✓ OK] OR PA RI SC SD TN [✓ TX] UT VA VT WA WI WV
WY

Select a commodity

ENERGY

Select a domain

WATER SOURCE

Select a data item

IN THE OPEN - ACRES IRRIGATED

Select domain categories

[✓ GROUND]
SURFACE

Select Years

[✓ 2013]
[✓ 2018]
[✓ 2023]

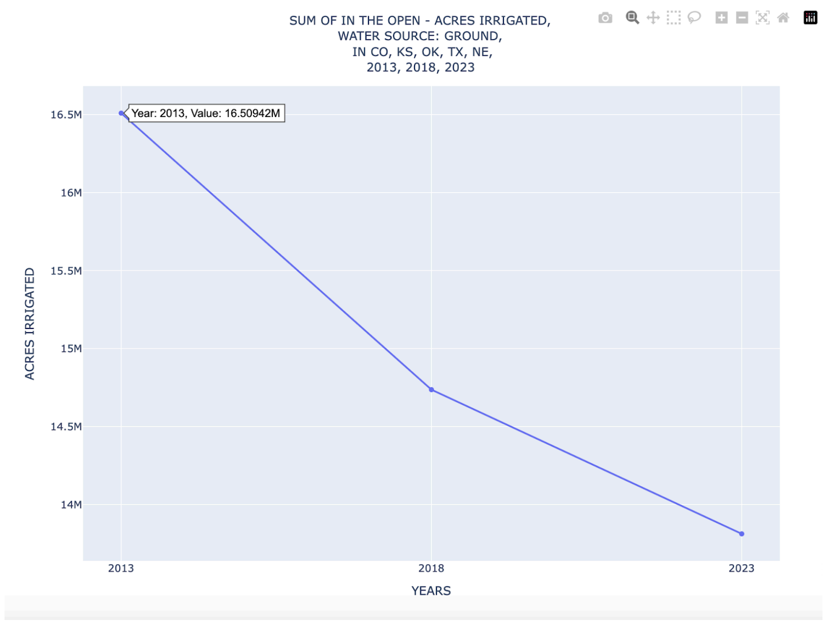
What statistic do you want to visualize/analyze?

Average [✓ Sum] Minimum Maximum

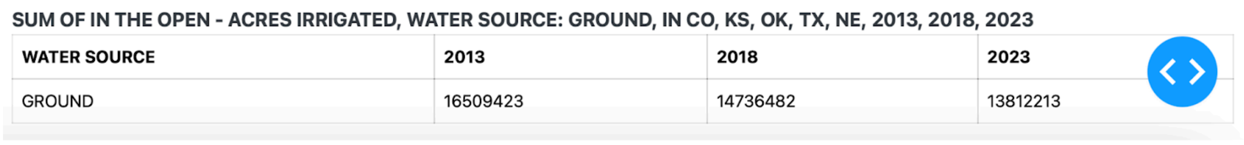
Do want one line, representing your operation over all states specified, or multiple lines, each representing your state and your specified operation for only that state?

Multiple Lines [✓ One Line]

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



Where the associated data table is:

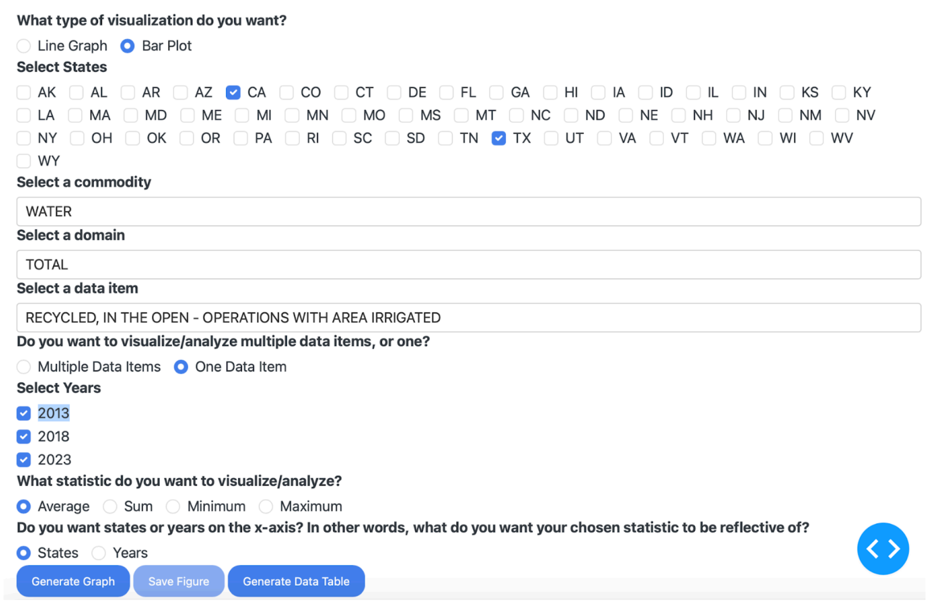


SUM OF IN THE OPEN - ACRES IRRIGATED, WATER SOURCE: GROUND, IN CO, KS, OK, TX, NE, 2013, 2018, 2023

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:



What type of visualization do you want?

Line Graph [✓ Bar Plot]

Select States

AK AL AR AZ [✓ CA] CO CT DE FL GA HI IA ID IL IN KS KY
LA MA MD ME MI MN MO MS MT NC ND NE NH NJ NM NV
NY OH OK OR PA RI SC SD TN [✓ TX] UT VA VT WA WI WV
WY

Select a commodity

WATER

Select a domain

TOTAL

Select a data item

RECYCLED, IN THE OPEN - OPERATIONS WITH AREA IRRIGATED

Do you want to visualize/analyze multiple data items, or one?

Multiple Data Items [✓ One Data Item]

Select Years

[✓ 2013]
[✓ 2018]
[✓ 2023]

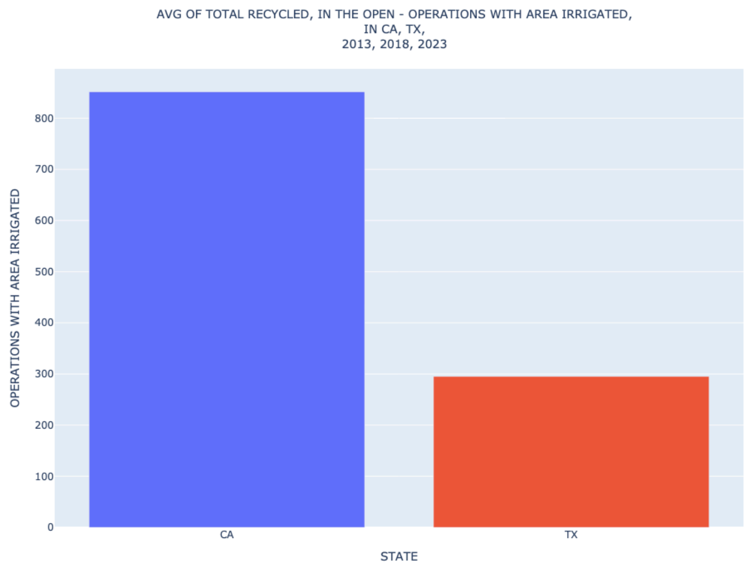
What statistic do you want to visualize/analyze?

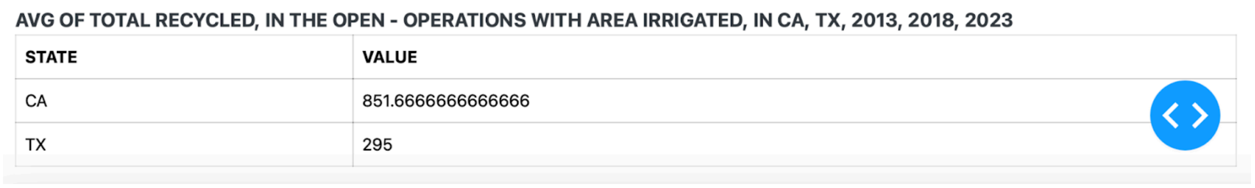
[✓ Average] Sum Minimum Maximum

Do you want states or years on the x-axis? In other words, what do you want your chosen statistic to be reflective of?

[✓ States] Years

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 prefer to irrigate with recycled water more 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 the graphs and data tables displayed in this section, 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 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 (the current setting). 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 saves them only as .png files. Ultimately, possible future work includes enhancing the tool further with regards to how users can visualize, save, and understand the data stored in the irrigation database.

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