

Stage V(b): User Interface (Group 4)

Group Members: Jason Kantner, Gordon Petry, Michael Williams, Casey Lishko, Yanaja Joyner, Len La Rocca

- For this stage, the database and code for the web interface are on the file homepage on each group member's VM
 - Folder Name: cab-civicstats-group-4
- All of the necessary code and documents is also on the group's GitHub repository
 - <https://github.com/gpetry37/CivicStats>

Stage V(a): Table and Queries (Group 4)

Group Members: Jason Kantner, Gordon Petry, Michael Williams, Casey Lishko, Yanaja Joyner, Len La Rocca

SQL Data Definition Queries

- Database, Tables, Constraints and View Creation Queries
 - Create database called **civstat** where all of the information will be stored
 - `createdb civstat`
 - Access PostgreSQL in the terminal to interact with the civstat database
 - `psql civstat`
 - Create **6 tables** that will be stored within civstat database, called Counties, Categories, Municipalities, Certification, Actions and Complete_Actions (In **create_tables.sql** file):
 - ```
CREATE TABLE Counties (
 CTYcode SERIAL NOT NULL,
 CTYname VARCHAR(30) NOT NULL,
 PRIMARY KEY (CTYcode)
);
```
    - ```
CREATE TABLE Categories (  
    Ccode SERIAL NOT NULL,  
    Cname VARCHAR(50) NOT NULL,  
    PRIMARY KEY (Ccode)  
);
```
 - ```
CREATE TABLE Municipalities (
 MUNcode SERIAL NOT NULL,
 MUNname VARCHAR(30) NOT NULL,
 PRIMARY KEY (MUNcode)
);
```

```

Mcode SERIAL NOT NULL,
Mname VARCHAR(30) NOT NULL,
CTYcode SERIAL NOT NULL,
PRIMARY KEY (Mcode),
FOREIGN KEY (CTYcode) REFERENCES Counties
(CTYcode) ON DELETE CASCADE
);

```

- CREATE TABLE Certification (

```

Mcode SERIAL NOT NULL,
Certification_Date DATE NOT NULL,
Total_Points INT NOT NULL,
Num_Gold_Stars INT NOT NULL,
Bronze_Silver VARCHAR(10) NOT NULL,
PRIMARY KEY (Mcode),
FOREIGN KEY (Mcode) REFERENCES Municipalities
(Mcode) ON DELETE CASCADE
);

```
- CREATE TABLE Actions(

```

Acode SERIAL NOT NULL UNIQUE,
Ccode SERIAL NOT NULL,
Aname VARCHAR(100) NOT NULL,
Point_Value INT NOT NULL,
Priority BOOLEAN NOT NULL,
Required BOOLEAN NOT NULL,
PRIMARY KEY (Acode, Ccode, Aname),
FOREIGN KEY (Ccode) REFERENCES Categories (Ccode)
ON DELETE CASCADE
);

```
- CREATE TABLE Complete\_Actions(

```

Acode SERIAL NOT NULL,
Mcode SERIAL NOT NULL,
PRIMARY KEY (Acode, Mcode),
FOREIGN KEY (Mcode) REFERENCES
Municipalities(Mcode) ON DELETE CASCADE,
FOREIGN KEY (Acode) REFERENCES Actions (Acode) ON
DELETE CASCADE
);

```

- The following queries will create **14 views** that will be stored in the civstat databases to serve various uses (In **create\_views.sql** file):

- 1. Shows the total points that could be earned by each category.

```
CREATE VIEW pt_tot_category AS
SELECT C.Cname, SUM(A.Point_Value) AS
Total_Points
FROM Actions AS A JOIN Categories AS C
ON A.Ccode = C.Ccode
GROUP BY C.Ccode
ORDER BY C.Cname;
```

- 2. Shows the total actions under each category.

```
CREATE VIEW tot_act_cat AS
SELECT C.Cname, COUNT(A.Acode) as Count_Action
FROM Actions AS A
JOIN Categories AS C
ON C.Ccode = A.Ccode
GROUP BY C.Ccode
ORDER BY C.Cname;
```

- 3. Shows the total actions each municipality did

```
CREATE VIEW tot_act_mun AS
SELECT M.Mname, COUNT(CA.Acode) AS Count_Action
FROM Complete_Actions AS CA JOIN Municipalities
AS M
ON CA.Mcode = M.Mcode
GROUP BY M.Mname
ORDER BY M.Mname;
```

- 4. Shows the total points each municipality received

```
CREATE VIEW tot_pts_mun AS
SELECT M.Mname, C.Total_Points
FROM Certification AS C JOIN Municipalities AS M
ON C.Mcode = M.Mcode
ORDER BY M.Mname;
```

- 5. Shows the total number of categories each municipality fulfilled

```
CREATE VIEW tot_cat_mun AS
SELECT M.Mname, COUNT(DISTINCT A.Ccode) AS
Count_Category
FROM Complete_Actions AS CA JOIN Actions A
ON CA.Acode = A.Acode
JOIN Municipalities AS M
ON CA.Mcode = M.Mcode
GROUP BY M.Mcode
```

```
ORDER BY M.Mname;
```

■ 6. Shows the total number of actions each county did

```
CREATE VIEW tot_act_cty AS
SELECT C.CTYname, COUNT(CA.Acode) AS Count_Action
FROM Municipalities AS M JOIN Complete_Actions AS
CA
ON M.Mcode = CA.Mcode
JOIN Counties AS C
ON C.CTYcode = M.CTYcode
GROUP BY C.CTYname
ORDER BY C.CTYname;
```

■ 7. Shows the total points each county received

```
CREATE VIEW tot_pts_cty AS
SELECT cy.CTYname, SUM(cert.Total_Points) AS
Total_Points
FROM Certification AS cert JOIN Municipalities AS
mun
ON cert.Mcode = mun.Mcode JOIN Counties as cy
ON cy.CTYcode = mun.CTYcode
GROUP BY mun.CTYcode, cy.CTYname
ORDER BY cy.CTYname;
```

■ 8. Shows the total number of categories each county fulfilled

```
CREATE VIEW tot_cat_cty AS
SELECT CY.CTYname, COUNT(DISTINCT A.Ccode)AS
Count_Category
FROM Municipalities AS M JOIN Complete_Actions AS
CA
ON M.Mcode = CA.Mcode
JOIN Actions A
ON CA.Acode = A.Acode
JOIN Counties as CY
ON CY.CTYcode = M.CTYcode
GROUP BY CY.CTYname
ORDER BY CY.CTYname;
```

■ 9. Shows the number of bronze stars per county

```
CREATE VIEW bronze_stars_cty AS
SELECT CY.CTYname, COUNT(C.Bronze_Silver) AS
Total_Bronze
FROM Certification AS C
```

```

JOIN Municipalities AS M
ON M.Mcode = C.Mcode
JOIN Counties AS CY
ON CY.CTYcode = M.CTYcode
WHERE Bronze_Silver = 'Bronze'
GROUP BY CY.CTYname
ORDER BY CY.CTYname;

```

- 10. Shows count silver stars per county.

```

CREATE VIEW silver_stars_cty AS
SELECT CY.CTYname, COUNT(
 CASE C.Bronze_Silver WHEN 'Silver'
 THEN 1 ELSE NULL END) AS Total_Silver
FROM Certification AS C
JOIN Municipalities AS M
ON M.Mcode = C.Mcode
JOIN Counties AS CY
ON CY.CTYcode = M.CTYcode
GROUP BY CY.CTYname
ORDER BY CY.CTYname;

```

- 11. Shows the number of non-priority actions per category

```

CREATE VIEW no_priority_cat AS
SELECT C.Cname, COUNT(A.Priority) AS
Total_NonPriority_Actions
FROM Actions AS A JOIN Categories AS C
ON C.Ccode = A.Ccode
WHERE Priority IS NOT True
GROUP BY Cname
ORDER BY Cname;

```

- 12. Shows the number of priority actions per category

```

CREATE VIEW yes_priority_cat AS
SELECT C.Cname, COUNT(
 CASE WHEN A.Priority IS True
 THEN 1 ELSE NULL END) AS
Total_Priority_Actions
FROM Actions AS A JOIN Categories AS C
ON C.Ccode = A.Ccode
GROUP BY Cname
ORDER BY Cname;

```

- 13. Shows the number of not required actions per category

```

CREATE VIEW no_required_cat AS
SELECT C.Cname, COUNT(A.Required) AS
Total_NonRequired_Actions
FROM Actions AS A JOIN Categories AS C
ON C.Ccode = A.Ccode
WHERE Required IS NOT True
GROUP BY Cname
ORDER BY Cname;

```

■ 14. Shows the number of required actions per category

```

CREATE VIEW yes_required_cat AS
SELECT C.Cname, COUNT(
CASE WHEN A.Required IS True
THEN 1 ELSE NULL END) AS
Total_Required_Actions
FROM Actions AS A JOIN Categories AS C
ON C.Ccode = A.Ccode
GROUP BY Cname
ORDER BY Cname;

```

■ Note: the **drop\_tables.sql** and **drop\_views.sql** files were used during the testing process

- Data Retrieval and Formatting Queries. Utilized through Web Scraper to retrieve necessary data. (Data is stored in the **StgV(a)\_Tables\_Group4 file**)
  - # Python 3 Web Scraper
  - 
  - # Ensure both beautifulsoup and requests are installed:
  - # pip install beautifulsoup4
  - # pip install requests
  - 
  - import requests
  - from bs4 import BeautifulSoup
  - 
  - map\_page = requests.get("https://www.sustainablejersey.com/certification/search-participating-municipalities-approved-actions/")
  - map\_data = map\_page.content
  - soup = BeautifulSoup(map\_data, "html.parser")
  - 
  - # lists all municipalities, listing them in the order of:
  - # Municipality, County, Certification

```

○ all = []
○
○ # finds all the tr tags in soup (the webpage) containing community=row as its
 class
○ for tr_tags in soup.find_all(attrs={"class": "community-row"}):
○ # finds all the td values in a given tr tag (there should be 3)
○ for td_tags in tr_tags.find_all("td"):
○ # Appends the values to all and removes whitespace
○ all.append(" ".join(td_tags.text.split()))
○ # print(all)
○ # total number should come out to 1365 with 455 municipalities
○ # print(len(all))
○
○ municipalities = []
○ counties = []
○ certifications = []
○ i = 0
○ for item in all:
○ if i % 3 == 0:
○ municipalities.append(item)
○ if i % 3 == 1:
○ counties.append(item)
○ if i % 3 == 2:
○ certifications.append(item)
○ i = i + 1
○
○ # print(municipalities)
○ # print(len(municipalities))
○
○ # Converts counties to a dictionary to remove duplicates
○ counties = list(dict.fromkeys(counties))
○ # print(counties)
○ # print(len(counties))
○
○ # Removes duplicates from certifications to show all certification values
○ certifications = list(dict.fromkeys(certifications))
○ # print(certifications)
○ # print(len(certifications))
○

```

```

○
○
○
○ # Start of Actions page
○
○ actions_page = requests.get("https://www.sustainablejersey.com/actions/")
○ actions_data = actions_page.content
○ soup2 = BeautifulSoup(actions_data, "html.parser")
○
○ # lists all categories
○ categories = []
○ # Lists all actions under each category. New categories are specified to make it
 easier to separate categories from actions
○ actions = []
○ # finds all the tr tags in soup2 (the webpage) containing community=row as its
 class
○ for li_tags in soup2.find_all(attrs={"class": "action-category"}):
○ # finds all the td values in a given tr tag (there should be 3)
○ i = 0
○ for h3_tags in li_tags.find_all("h3"):
○ # Appends the values to all and removes whitespace
○ categories.append(" ".join(h3_tags.text.split()))
○ actions.append("NEW_CATEGORY")
○ actions.append(" ".join(h3_tags.text.split()))
○ for sub_cat in li_tags.find_all(attrs={"class":
"action-subcategory--header"}):
○ actions.append("NEW_SUB_CAT")
○ actions.append(" ".join(sub_cat.text.split()))
○ # Actions are all printed in order
○ for h4_tags in li_tags.find_all("h4"):
○ actions.append(" ".join(h4_tags.text.split()))
○ # Points are all printed in order
○ for div_tags in li_tags.find_all(attrs={"class": "action--points"}):
○ actions.append(" ".join(div_tags.text.split()))
○ for req_tags in li_tags.find_all(attrs={"class": "action--req"}):
○ actions.append(" ".join(req_tags.text.split()))
○
○ # print(categories)
○

```



- # The category is first in the list, followed by a sub category (specified by NEW\_SUB\_CAT),
- # followed by all of the actions, followed by their corresponding point values, followed by
- # the items that are priority or required.
- print(actions)
- Table Population Queries
  - There are **6 SQL files** submitted that contained the insert queries that inserted the data into the table. These files are called **insert\_county.sql, insert\_category.sql, insert\_municipality.sql, insert\_certification.sql, insert\_action.sql, insert\_complete\_action.sql.**
  - In the VM, the files go through the psql -f statement to store the data into the civstat database.

### ***SQL Data Manipulation Queries***

- SQL Data Definition Queries
  - These queries are stored in the **manipulation\_queries.sql** file that was submitted
  - These queries are an example of manipulation queries that will be utilized in the database. More may be added later in the process and it will be updated appropriately.
- The database is within each group member's Virtual Machine
  - The schema is currently present in the VM
  - To access the files in the VM, it is located below:
    - Home → cab-civicstats-group-4

### Stage IV: Design (Group 4)

Group Members: Jason Kantner, Gordon Petry, Michael Williams, Casey Lishko, Yanaja Joyner, Len La Rocca

#### *Boyce-Codd Normal Form (BCNF) for Relations*

- **Conditions that must be Satisfied for a relation to be in BCNF:**
  - **1. Must be in 3NF**
  - **2. For any dependency  $A \rightarrow B$ , A should be a super key**

- Counties

- Attributes: *CTYcode*, *CTYname*
- The County relation is normalized to Boyce-Codd Normal Form.
  - This relation is normalized to BCNF because it satisfies both conditions. It satisfies the conditions for all the required forms:
    - 1NF: Values are atomic, each value in a given column is of the same type, each column has a unique name and the order of insertion does not matter.
    - 2NF: Satisfies 1NF and has no partial dependencies. There is only one prime attribute, so partial dependency is not an issue.
    - 3NF: Satisfies 2NF and has no transitive dependency. The non-prime attribute *CTYname* is not dependent on another non-prime attribute.
  - Since all of these conditions are satisfied, that means that condition 1 for BCNF is also satisfied. Condition 2 is satisfied because there are no violations. The prime attribute of the relation, *CTYcode*, is not dependent on a non-prime attribute.

- Municipalities

- Attributes: *Mcode*, *Mname*, *CTYcode*, *Certification\_Date*, *Total\_Points*, *Certified*, *Num\_Gold\_Stars*, *Bronze\_Silver*
  - Foreign Keys: *CTYcode*
- The Municipalities relation is not normalized to Boyce-Codd Normal Form
  - This relation is not normalized to BCNF because it violates the first condition. This relation is not in 3NF because there are transitive dependencies present. The non-prime attributes *Certification\_Date*, *Total\_Points*, *Num\_Gold\_Stars*, *Bronze\_Silver* are all dependent on the non-prime attribute, *Certified*. However, condition 2 is satisfied because no prime attributes (*Mcode*) are dependent on non-prime attributes.
- To normalize this relation into BCNF, it must be decomposed into two relations as listed below
  - **Municipalities:** *Mcode*, *Mname*, *CTYcode*
    - Foreign Keys: *CTYcode*
  - **Certification:** *Mcode*, *Certification\_Date*, *Total\_Points*, *Num\_Gold\_Stars*, *Bronze\_Silver*
    - Foreign Key: *Mcode*
- With both of these relations, there are no longer any transitive dependencies because the non-prime attributes of each relation are only dependent on their respective prime attributes. Also, we decided to cut the *Certified* attribute because all of the municipalities we are storing are certified. Although we did not have

time to implement, we would have liked to have non-certified municipalities as well. With this change, condition 1 is now satisfied along with condition 2, meaning that these relations are both normalized in BCNF.

- Actions

- Attributes: Acode, Mcode, Ccode, *Aname*, *Point\_Value*, *Priority*, *Required*
  - *Acode*, *Mcode* and *Ccode* form a composite key
  - Foreign Keys: *Mcode* and *Ccode*
- The Actions relation is not normalized to Boyce-Codd Normal Form
  - This relation violates condition 1 because it is not in 2NF, meaning it cannot be in 3NF. This relation is not in 2NF because there are partial dependencies
    - *Acode* -> *Aname*
  - To normalize this relation into BCNF, it must be decomposed into two relations as listed below:
    - **Actions:** Acode, Ccode, *Aname*, *Point\_Value*, *Priority*, *Required*
    - **Completed Actions:** Acode, Mcode.
- With these new relations, there are no longer any partial dependencies because each non-prime attribute depends on each prime attribute in the relation. There is also no transitive dependency which fulfils condition one. Finally, the prime attributes are not derived from non-prime attributes thus fulfilling condition 2, making this new relation normalized in BCNF.

- Categories

- Attributes: Ccode, *Cname*
- The Categories relation is normalized to Boyce-Codd Normal Form
  - This relation is normalized to BCNF because it satisfies both conditions. It satisfies the conditions for all the required forms:
    - 1NF: Values are atomic, each value in a given column is of the same type, each column has a unique name and the order of insertion does not matter
    - 2NF: Satisfies 1NF and has no partial dependencies. There are no partial dependencies because there is only one prime attribute.
    - 3NF: Satisfies 2NF and has no transitive dependency. The non-prime attribute *Cname* is not dependent on another non-prime attribute.
  - Since all of these conditions are satisfied, that makes condition 1 of BCNF satisfied as well. Condition 2 is satisfied because no prime attributes are dependent on non-prime attributes. The only prime attribute of this relation, *Ccode* is not dependent on a non-prime attribute.

## *Views*

- Define the different views required. For each view list the data and transaction requirements. Give a few examples of queries, in English, to illustrate.
- What do we want users to be able to display based on the data in the database? (Views Required)
- Each will be put in the format of (X axis) v (Y axis) for a bar graph, where a pie chart the X axis will correlate to the colored item, and the total will correlate to the size of each section.
  - 1 Specific Municipality
    - Municipality v Total points
    - Municipality v Total actions
    - Municipality v Total categories contributed to
  - N Municipalities
    - Municipalities v Total points
    - Municipalities v Total actions
    - Municipalities v Total categories contributed to
  - All Municipalities
    - Municipalities v Total points
    - Municipalities v Total actions
    - Municipalities v Total categories contributed to
  - 1 Specific County
    - County v Total points
    - County v Total actions
    - County v Total categories contributed to
    - County v # of bronze municipalities
    - County v # of silver municipalities
  - N Counties
    - Counties v Total points
    - Counties v Total actions
    - Counties v Total categories contributed to
    - Counties v # of bronze municipalities
    - Counties v # of silver municipalities
  - All Counties
    - Counties v Total points
    - Counties v Total actions
    - Counties v Total categories contributed to
    - Counties v # of bronze municipalities
    - Counties v # of silver municipalities

- 1 Specific Category
  - Category v Total points
  - Category v Total actions
  - Category v # priority actions
  - Category v # non-priority actions
  - Category v # required actions
  - Category v # non-required actions
- N Categories
  - Categories v Total points
  - Categories v Total actions
  - Categories v # priority actions
  - Categories v # non-priority actions
  - Categories v # required actions
  - Categories v # non-required actions
- All Categories
  - Categories v Total points
  - Categories v Total actions
  - Categories v # priority actions
  - Categories v # non-priority actions
  - Categories v # required actions
  - Categories v # non-required actions

### Queries

- Design a complete set of queries to satisfy the transaction requirements identified in the previous stages.

These queries are written out as a mix of sql and english. Parentheses are used to make joins easier to read

- 1 specific municipality
  - Municipality v Point total
    - X: SELECT Mname FROM Municipality WHERE Mname = \*user input\*
    - Y: SELECT Total\_Point FROM (Municipality INNER JOIN Participating Municipality ON Mcode = Mcode) WHERE Mname = \*user input\*
  - Municipality v Total actions
    - X: SELECT Mname FROM Municipality WHERE Mname = \*user input\*
    - Y: SELECT count(\*) FROM (Municipality INNER JOIN Action ON Mcode = Mcode) WHERE Mname = \*user input\*
  - Municipality v categories contributed to

- X: SELECT Mname FROM Municipality WHERE Mname = \*user input\*
  - Y: SELECT count(Cname) FROM (Municipality INNER JOIN Action ON Mcode = Mcode INNER JOIN Category ON Ccode = Ccode) WHERE Mname = \*user input\*
- N municipalities
  - Municipalities v Point total
    - X: SELECT Mname FROM Municipality WHERE Mname = \*user input\* AND Mname = \*user input 2\* AND ... Mname = \*user input N\*
    - Y: SELECT Total\_Point FROM (Municipality INNER JOIN Participating Municipality ON Mcode = Mcode) WHERE Mname = \*user input\* AND Mname = \*user input 2\* AND ... Mname = \*user input N\*
  - Municipalities v Total actions
    - X: SELECT Mname FROM Municipality WHERE Mname = \*user input\*
    - Y: SELECT count(\*) FROM (Municipality INNER JOIN Action ON Mcode = Mcode) WHERE Mname = \*user input\* AND Mname = \*user input 2\* AND ... Mname = \*user input N\*
  - Municipalities v categories contributed to
    - X: SELECT Mname FROM Municipality WHERE Mname = \*user input\* AND Mname = \*user input 2\* AND ... Mname = \*user input N\*
    - Y: SELECT count(Cname) FROM (Municipality INNER JOIN Action ON Mcode = Mcode INNER JOIN Category ON Ccode = Ccode) WHERE Mname = \*user input\* AND Mname = \*user input 2\* AND ... Mname = \*user input N\*
- ALL Municipalities
  - Municipalities v Point total
    - X: SELECT Mname FROM Municipality
    - Y: SELECT Total\_Point FROM (Municipality INNER JOIN Participating Municipalities ON Mcode = Mcode)
  - Municipalities v Total actions
    - X: SELECT Mname FROM Municipality
    - Y: SELECT count(\*) FROM (Municipality INNER JOIN Action ON Mcode = Mcode)
  - Municipalities v categories contributed to
    - X: SELECT Mname FROM Municipality
    - Y: SELECT count(Cname) FROM (Municipality INNER JOIN Action ON Mcode = Mcode INNER JOIN Category ON Ccode = Ccode)

- 1 Specific County
  - County v Point Total
    - X Axis: CTYname
    - Y Axis: SELECT Total\_Point FROM ((Municipalities INNER JOIN Certification ON Mcode) INNER JOIN Counties ON CTY Code Where CTY Code = \*user input\*
  - County v # of gold actions
    - X Axis: CTYname
    - Y Axis: SELECT SUM(Num\_Gold\_Stars) FROM ((Municipalities INNER JOIN County ON CTYcode) INNER JOIN Certifications ON Mcode) WHERE CTY Code = \*user input\*
  - County v # of silver actions
    - X Axis: CTYname
    - Y Axis: SELECT Count(Mcode) FROM ((Municipalities INNER JOIN County ON CTYcode) INNER JOIN Certifications ON Mcode) WHERE CTY Code = \*user input\* AND Bronze\_Silver = true
  - County v Total Categories
    - X Axis: CTYname
    - Y Axis: COUNT Ccode FROM (Natural Join Counties, Municipalities, Complete Actions, Actions, Categories) WHERE CTYcode = \*user input\*
  - County v Total Actions
    - X Axis: CTYname
    - Y Axis: COUNT Acode FROM (Municipalities INNER JOIN County ON CTYcode) INNER JOIN Actions WHERE CTYcode = \*user input\*
  
- N Counties and ALL Counties
  - County v Point Total
    - X Axis: CTYname
    - Y Axis: SELECT Total\_Point FROM ((Municipalities INNER JOIN Certification ON Mcode) INNER JOIN Counties ON CTY Code Where CTY Code = \*user input\* AND CTYcode = \*user input 2\*
  - County v # of gold actions
    - X Axis: CTYname
    - Y Axis: SELECT SUM(Num\_Gold\_Stars) FROM ((Municipalities INNER JOIN County ON CTYcode) INNER JOIN Certifications ON Mcode) WHERE CTY Code = \*user input\* AND CTYcode = \*user input 2\*
  - County v # of silver actions
    - X Axis: CTYname

- Y Axis: SELECT Count(Mcode) FROM ((Municipalities INNER JOIN County ON CTYcode) INNER JOIN Certifications ON Mcode) WHERE CTY Code = \*user input\* AND CTYcode = \*user input 2\* AND Bronze\_Silver = true
  - County v Total Categories
    - X Axis: CTYname
    - Y Axis: COUNT Ccode FROM (Natural Join Counties, Municipalities, Complete Actions, Actions, Categories) WHERE CTYcode = \*user input\* AND CTYcode = \*user input 2\*
  - County v Total Actions
    - X Axis: CTYname
    - Y Axis: COUNT Acode FROM (Municipalities INNER JOIN County ON CTYcode) INNER JOIN Actions WHERE CTYcode = \*user input\* AND CTYcode = \*user input 2\*
- 1 Category
  - Category v Total points
    - X Axis: Cname
    - Y Axis: SELECT SUM(Total\_Points) FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\*
  - Category v Total actions
    - X Axis: Cname
    - Y Axis: COUNT Acode FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\*
  - Category v # priority actions
    - X Axis: Cname
    - Y Axis: COUNT Acode FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\* AND Priority = True
  - Category v # non-priority actions
    - X Axis: Cname
    - Y Axis: COUNT Acode FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\* AND Priority = False
  - Category v # required actions
    - X Axis: Cname
    - Y Axis: COUNT Acode FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\* AND Required = True
  - Category v # non-required actions
    - X Axis: Cname
    - Y Axis: COUNT Acode FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\* AND Required = False



- N and ALL Categories
  - Categories v Total points
    - X Axis: Cname
    - Y Axis: SELECT SUM(Total\_Points) FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\* AND Cname = \*user input 2\*
  - Categories v Total actions
    - X Axis: Cname
    - Y Axis: COUNT Acode FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\* AND Cname = \*user input 2\*
  - Categories v # priority actions
    - X Axis: Cname
    - Y Axis: COUNT Acode FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\* AND Cname = \*user input 2\* AND Priority = True
  - Categories v # non-priority actions
    - X Axis: Cname
    - Y Axis: COUNT Acode FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\* AND Cname = \*user input 2\* AND Priority = False
  - Categories v # required actions
    - X Axis: Cname
    - Y Axis: COUNT Acode FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\* AND Cname = \*user input 2\* AND Required = True
  - Categories v # non-required actions
    - X Axis: Cname
    - Y Axis: COUNT Acode FROM Actions INNER JOIN Categories ON Ccode WHERE Cname = \*user input\* AND Cname = \*user input 2\* AND Required = False

---

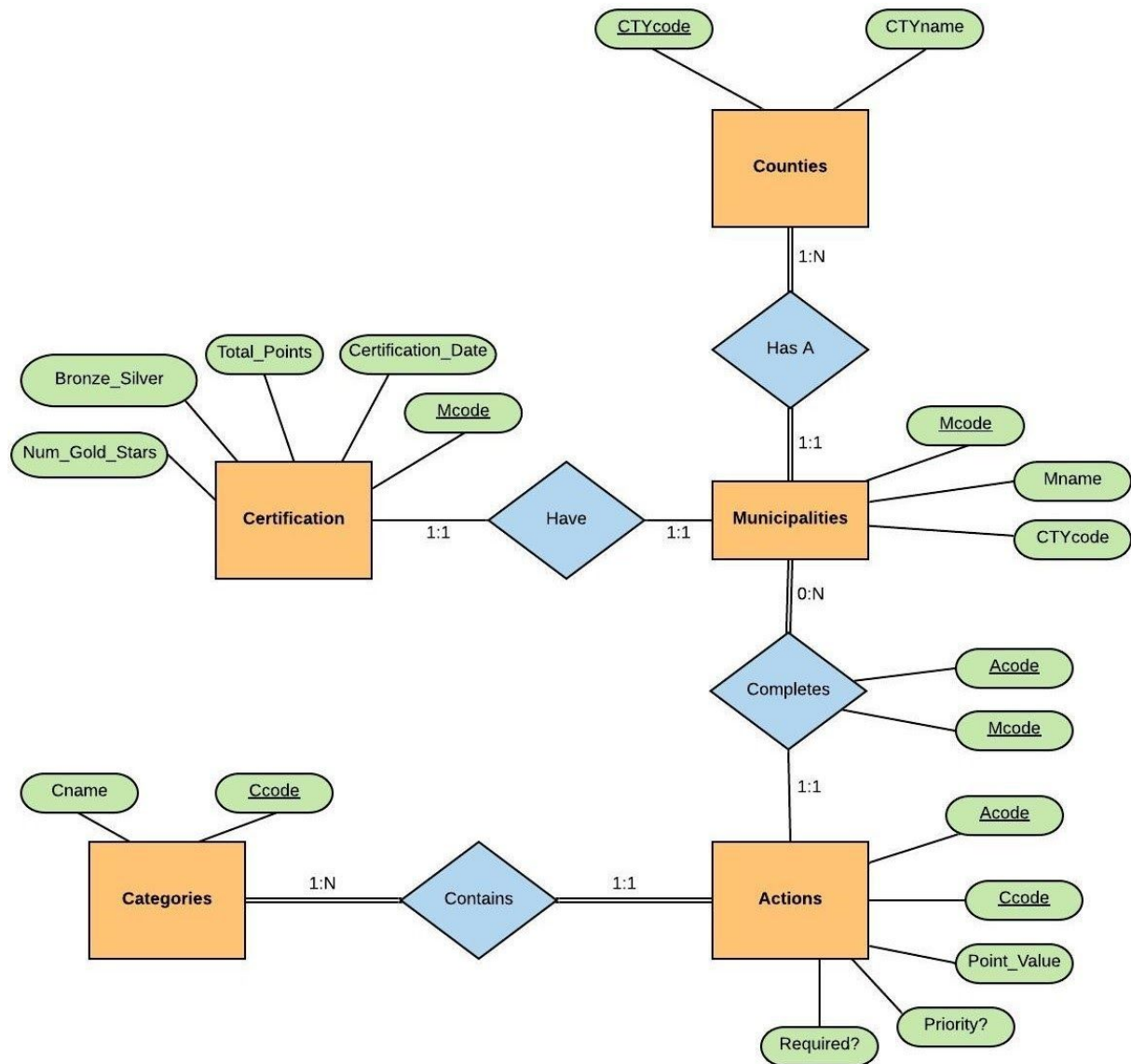
### Stage III: Data Model (Group 4)

Group Members: Jason Kantner, Gordon Petry, Michael Williams, Casey Lishko, Yanaja Joyner, Len La Rocca

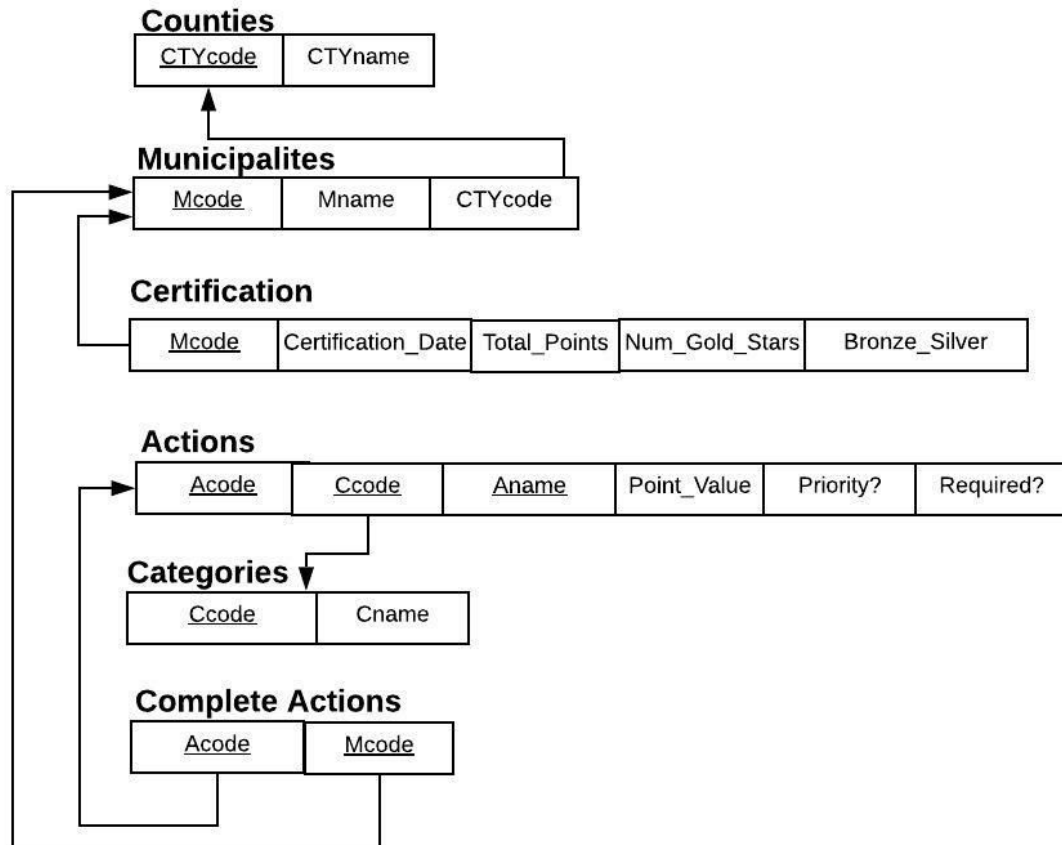
Link to LucidChart:

[https://www.lucidchart.com/documents/edit/897a80eb-bf1d-41a9-9c7d-81e4dfc2bb1c/0\\_0?share\\_d=true](https://www.lucidchart.com/documents/edit/897a80eb-bf1d-41a9-9c7d-81e4dfc2bb1c/0_0?share_d=true)

## ER Diagram



## ***Relational Schema***



## ***Additional Information***

- Initial Database Size (approximate number of records)
  - Our initial database hold **1,780** records
    - Counties: 21
    - Municipalities: 455
    - Certification: 61
    - Actions: 156

- Categories: 18
  - Complete Actions: 1069
  - Types and average number of searches
    - Our initial estimate of queries that will be used in a typical user session is around **2 queries**
      - We got this estimate based on the following actions that must be taken
        - 1. Selecting a tab/page to pick an X axis
        - 2. Selecting the specific rows the user wants to see on the X axis and picking an Y axis from the drop down menu
      - When the “Submit Query” button is hit, the 2 generated queries based on the user’s selection will be run and a bar chart will be generated.
- 

## Stage II: Project Proposal and Specifications (Group 4)

Group Members: Jason Kantner, Gordon Petry, Michael Williams, Casey Lishko, Yanaja Joyner, Len La Rocca

Link to GitHub Wiki: <https://github.com/CSC-315/cab-civicstats-group-4>

### **Problem Statement**

For this project, the main problem that we want to address and find a solution for is the lack of easily accessible/understandable sustainability data on the CivicStory website. Currently, the CivicStory website is a hub for relevant news articles relating to sustainability, however direct information on sustainability efforts based on geographic location is not accessible to the common user.

### **Objective of the Module**

The objective of this module is to create an addition to the CivicStory website that will allow normal users to access detailed data on sustainability in their area. Another objective which we hope to accomplish is to ensure that this data is easily understandable to the common user and it is tailored to what information they want to know. It is our hope that this tool will keep the public well informed about sustainability in their own area.

### **Desired End Product / Part Developed for This Class**

In general, the product we plan to design can be used by many different websites and is not only limited to CivicStory. The module that will be designed will be a page where the user can choose from many different variables relating to data in a connected database. Once the user

specifies the data they want to see, queries will pull data from the connecting database in PostgreSQL and the desired data will be visualized onto the screen of the user. This data will be visualized as a bar graph. For this class, the data in the database will be relating to sustainability in the New Jersey area. This makes the planned module a great tool to be utilized on the CivicStory website because it strongly relates to the purpose of the website and the goals it wants to accomplish.

### **Importance of this Module and How it Addresses the Problem**

Right now CivicStory is mostly a stream of news articles relating to sustainability efforts. Although informative, these do not offer the user a look at raw data that relates to the issue. Allowing the user to search for specific data gives them a true understanding of the issues revolving around sustainability and what can be done to improve it in the future. Also, they can compare and contrast data from different times or places to get the “whole picture” of where sustainability is, where it comes from, and where it needs to go.

### **Problem Domain Research Plan and Data Obtainment**

To research the problem domain, we will look over the CivicStory website to see what type of data would be relevant to put in the database connected to the website. Not every piece of sustainability data we find should be put in the database. For example, we need to have a limited geographic location so it remains relevant to those that will access the CivicStory website. Once we research the problem domain in more depth, it will then be much easier to figure out what kind of data we will implement into our database. We obtained our data from the Sustainable Jersey website. This website has a large amount of data regarding sustainability in New Jersey divided by counties. We will collect this data and store it in our PostgreSQL database.

### **Similar Systems and How our System Differs**

Search bars and user interfaces that allow for specific search terms do exist in many forms (Ex. PAWS class search tool). However, the addition that we plan to make ours stand out from others in that our search tool creates a data visualization based on the information provided by the user on what data they want to view.

### **Other Possible Applications for this System**

The search system has wide applications on many databases. It is a robust search system that visualizes the data, which is useful in any database. Possibly, instead of typing in the search criteria, the user can point out on a map where they want to search, or use a slider to determine what time period they want to look at. However, the data visualization tool will always be used after the search.

### **Performance**

The system we plan to design will ensure efficiency between the many programs that are being utilized. Our goal is to reduce lag time as much as possible between the Database Management System and the web interface. This way, when the user prompts to create a data visualization, the task will be completed in a reasonable amount of time. Performance and efficiency should always be a top priority when creating a new web tool to use. This will be accomplished by taking time and planning out our code before going into the programs right away. We will also continue to test our system throughout the development process to see what areas we need to improve.

### **Security**

Regarding the security of our system, we don't plan to allow users to directly alter the data within the PostgreSQL database. This access will be limited to only administrators (our group members), who are allowed to do so. For the users, they will only be able to select the parameters in which they want to query the data and view the visualization that it provides. So, the users are only limited to viewing the web interface side of this module.

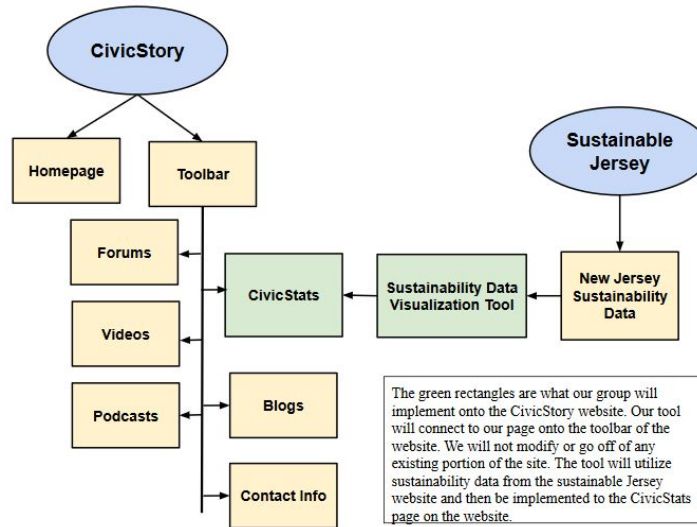
### **Backup and Recovery**

Backup and recovery are an important part when implementing a tool that utilizes data. For our project, we plan to have a copy of the database in each of the group member's virtual machines. We will also have copies of the required files in both a shared Google Drive and a GitHub Repository. This means that if anything happens to the database connected to the website, we will have a copy of the data just in case it needs to be utilized.

### **Technology and Database Concepts to Learn**

For this project, there are a few database concepts and technologies that our group will need to learn to be able to accomplish our objective. First, our team will need to learn how to utilize the Database Management System PostgreSQL. It is within PostgreSQL where our database containing the sustainability data will be located. We will also utilize HTML to create the web interface side of this project. We will use the software Flask to produce the web page where our information will be shown. This will create the frontend of the project where the user will view the data visualization tool. To implement this data visualization tool, we will also learn how to use data visualization software. The data visualization software we focused on is Chart.js which is a Javascript package that will generate a chart on the Flask webpage. Python will also have to be learned and utilized to bridge the gap between the web interface and the Database Management System. Python is very important to learn because without it, we would not be able to make the connection between the front and back end portions of this module.

### ***Diagrammatic Representation***



## Quad Chart



### Sustainability Data Visualization Tool

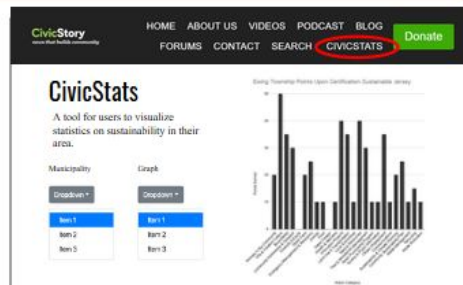
Jason Kantner, Gordon Petry, Michael Williams, Casey Lishko, Yanaja Joyner, Len La Rocca

#### Objective

Develop a system that converts sustainability data into easily understood visualizations.

CivicStory does not currently have resources on sustainability data and this will allow users to choose what data they want to see and in what form.

We hope to inform common users about sustainability and how it impacts their daily lives.



#### Approach

Modify the existing CivicStory website by

1. Creating data visualization software that will allow users to see specific data represented in a graph, pie chart, etc
2. Creating drop down menus that allow the user to select what type of data is visualized and in what form (graph type) it is shown.
3. Utilize tools such as PostgreSQL and Python to streamline the software so it can run efficiently and in a timely manner.

#### Key Milestones

- Proposal & Pitch ----- 2/12/20
- Specifications ----- 2/24/20
- Model ----- 3/09/20
- Design ----- 3/26/20
- Tables and Queries ----- 4/09/20 in PostgreSQL
- Implementation ----- 4/27/20
- Demos & Presentations ----- 5/04/20
- Final Report ----- 5/04/20