Table of Contents

Introduction

Members: Serena Lynas sjl132, Ethan Kaji tak112, Angel Diaz sad123 Project Team 22

Demo Date: 4/23 @ 12PM

Title: Campaign Manager

DB Name: CampaignManager

Abstract:
During the course of a political campaign, politicians need to manage a long list of contacts, host events and invite potential voters to those events, and collect donations. Additionally, campaign managers will want to be able to analyze their voters at scale in order to leverage insights about their base. This database will track all of this data, in addition to a voter's issues of interest and their voting history, in order to fine-tune political messaging, host relevant events, find likely donors, and ultimately run a successful campaign.

Reproduction Instructions

DB Backup Name - CampaignManager.bak

Zip File Name (Java Code) - CampaignManager.zip

Demo Schedule Date: 4/23 @ 12PM

How to Access Virtual Environment

Serena Lynas sjl132

SQL Server Name - cxp-sql-03
Database Name - CampaignManager
SA password - x62fuuVQvEvnu1
The database user name "sa" with password "x62fuuVQvEvnu1"

Compiling

Open the provided zip in the VSCode in the VM, then use the Java extension to run the project in the terminal. (Alternatively, compile manually with `javac` and add the required libs to the classpath).

Database Design and Business Rules

Relational Schema:

person(person_id, first, last, dob, phone, email, address, district)
event(event_id, venue_id, campaign_id, max_capacity. name, description)
donation(donation_id, person_id, campaign_id, amount)
venue(venue_id, name, location, max_capacity)
issue(issue_id, description)
campaign(campaign_id, candidate_id, manager_id, election_id, funds)
vote_cast(person_id, election_id, type, timestamp, voted_for)
election(election_id, registration_deadline, date)
attend_event(event_id, person_id, register_utm, register_timestamp)
person_issue(person_id, issue_id)

Relational Schema Description:

Person:

The `person` relation contains all people registered into the database. This includes candidates and campaign managers.

- `person_id` is a an integer primary key which uniquely identifies a person
- `first` is the person's first name
- `last` is the person's last name
- `dob` is the person's date of birth
- `phone` is the person's phone number
- `email` is the person's email address
- `address` is the person's mailing address
- `district` is a string that represents the district that the person lives in, e.g. `"OH-11"` for Ohio's 11th district in the U.S. House of Representatives.

Event:

The `event` relation contains all political events which guests are invited to (e.g. political rallies, dinners, etc.).

- `event_id` is an integer primary key which uniquely identifies the event.
- `venue_id` is a foreign key to the venue (i.e. location) that the event is hosted at. If the event is online or over the phone, this is null.
- `campaign_id` is a foreign key to the campaign that this event belongs to.

- `max_capacity` is the maximum number of attendees. Null if there is no limit (e.g. for an online event)
- `name` is a user-friendly name for an event (e.g. “Freedom Rally”)
- `description` is a user-friendly description of the event.
- `time_start` is the start time of the event as milliseconds from Unix epoch.
- `time_end` is the end time of the event as milliseconds from the Unix epoch.
- `type` is the type of the event, one of “town hall”, “gotv”, “rally”, “phone bank”, “other”

Donation:

The `donation` relation contains all donations from people to campaigns and the respective amounts.

- *`donation_id`* is an integer primary key assigned to each individual donation.
- *`person_id`* is a foreign key to the person who donated the amount
- *`campaign_id`* is a foreign key to the campaign the donation was going towards
- *`amount`* is the amount of money that was donated

Venue:

The `venue` relation contains all possible locations for events held by campaigns.

- `venue_id` is an integer primary key which uniquely identifies a venue
- `name` is the name of the venue
- `location` is the location of the venue
- `max_capacity` is the maximum number of people the venue can support for events hosted in the venue.

Issue:

The `issue` relation contains the various issues that individual persons care about. The issues relation can be used to identify the platform a campaign might run on.

- `issue_id` is an integer primary key which uniquely identifies an issue
- `description` is a description of the issue

Campaign:

The `campaign` relation describes all political campaigns registered within the database.

- `campaign_id` is the an integer primary key which uniquely identifies a campaign
- `candidate_id` is a foreign key to the person who is the campaign’s candidate
- `manager_id` is a foreign key to the person who is the campaign’s manager

- `election_id` is a foreign key to the election the campaign ran in
- `funds` tracks the current funds available to a given campaign

Vote Cast:

The `vote_cast` relation represents the voting history of a given person in the database.

- `person_id` is a foreign key to the person who is casting the vote
- `election_id` is a foreign key to the election a person voted in
- `type` is a string representing the type of vote cast (e.g. “early”, “mail in”, …)
- `timestamp` is the timestamp at which the vote was received
- `voted_for` is a possibly null foreign key representing which campaign the person voted for in the election if it is known

Election:

The `election` relation contains information on all the elections within the database.

- `election_id` is the integer primary key assigned to each election that occurs
- `registration_deadline` is the date that registration for the election ends
- `date` is the date that the election occurs

Attend Event:

The `attend_event` relation represents the attendance record of people at various events.

- `event_id` is a foreign key to the event the person attended
- `person_id` is a foreign key to the person who attended the event
- `register_utm` is
- `register_timestamp` is the timestamp at which the person registered for an event

Person Issue:

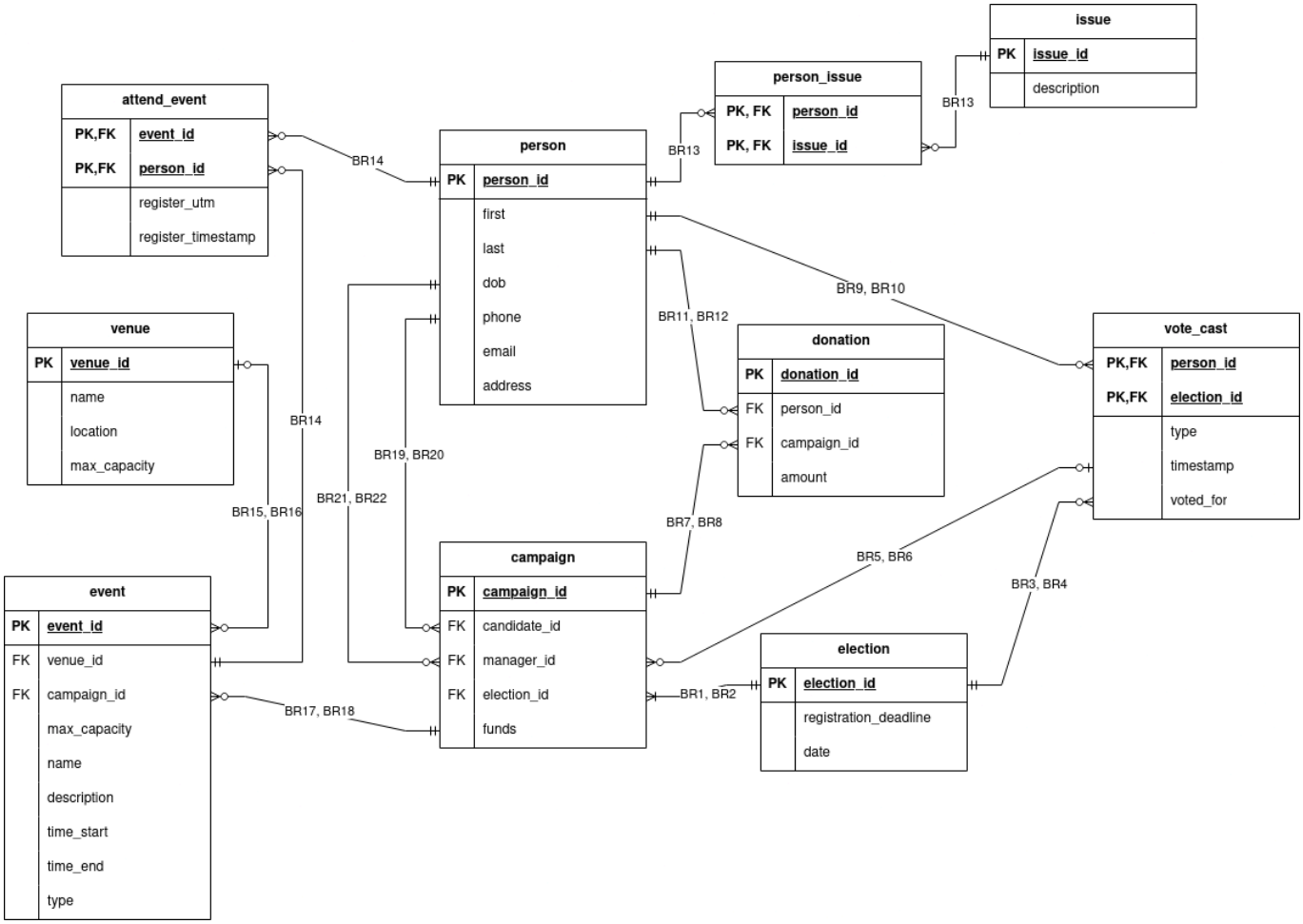
The attributes in person_issue correspond to identifiers from the person and issue relations and are named accordingly.

Business Rules:

BR1: An election must have at least one campaign
BR2: A campaign must occur in exactly one election
BR3: An election can have any number of votes cast in it
BR4: A vote can only be cast in one election
BR5: A vote can only be cast for one campaign

BR6: A campaign can have any number of votes cast for it
BR7: A donation can only be made to one campaign
BR8: A campaign can receive many or no donations
BR9: A person can cast any number of votes
BR10: A vote must be associated with just one person
BR11: A person can make any number of donations
BR12: A donation must be made by a single person
BR13: A person can be interested in any number of issues
BR14: A person can attend any number of events
BR15: A venue can host any number of events
BR16: An event does not need a venue
BR17: A campaign can host any number of events
BR18: An event must be hosted by a campaign
BR19: A person can be a candidate for any campaigns
BR20: A campaign must have a candidate
BR21: A person can be a campaign manager for any campaigns
BR22: A campaign must have a campaign manager

ERD:



Use Cases

Usage by the General Public

Use cases

1. People looking to find events, such as information sessions to attend (Serena Lynas)

Description: Members of the general public often want to attend events with a candidate, such as a town hall where they can ask questions to the candidate, voice their concerns, and hear how the candidate will address those concerns once elected (and ultimately why they should vote for them). Using our application, these users would be able to search for, find, and sign up for events with their local candidate.

Details: This use case serves the general public who want to be able to meet with their local candidates. The user will first log on with their ID (through `get_person`, which selects from the

person table), then the application will find their local candidates based on their ID (using the stored procedure find_local_candidates, which is a SELECT on a join of campaign, person, and election, in order to find the local candidates), and it will prompt the user for which of their local candidates they would like to attend events for. Then, the user will be prompted to filter by event type if they wish, for example, only showing town halls if that is the only thing the user is interested in. Lastly, the user will be shown the events, which are gotten through query_events, which uses a SELECT on event joined with campaign and person in order to find events that match the chosen candidate in the person table. If the type_filter parameter is provided, the results will further be filtered by what types appear in the filter string, where I used the STRING_SPLIT function to split the string into a small table. We will also want to prevent showing events which have already occurred (no point in signing up for an event in the past), so we filter by events with an ending time greater than SYSUTCDATETIME() to ensure that the event hasn't already ended. Lastly, the user will be prompted to sign up for events which interest them, through register_for_event, which will first verify that the event is not at capacity. I used a transaction here because we need atomicity: if two people register for an event at the same time, we want to ensure that there is no data races with COUNT(*) – we want it so that after COUNT(*) is checked, it isn't updated until after user is registered for the event, so I use a transaction to achieve this atomicity. After using SELECT on attend_event to calculate the number of attendees, we check that the sum plus one would still be under the max capacity and then if so, add the user to the attendees list with an INSERT INTO statement on the attend_event table. (2 DML used – SELECT and INSERT).

This use case uses the person_by_district index as it is querying candidates by district (this index is also used by another use case), the reason being that seeing the people/candidates who live in a district is a common process, and we want to avoid reading the entire person table – instead, we can just read the part that contains our district, using this index.

User requirements: The user must know their own User ID in order to be able to log into the system. The system will query the database for local candidates based on that user ID, so no other information is needed; everything else will be prompted. (Valid user ID's include 1 through 10). The system may return nothing if there are no events which match the given filters, just try a different district/event types to find something.

SQL Code

```
Unset
CREATE INDEX person_by_distict ON person(district);

GO
CREATE OR ALTER PROCEDURE get_person
```

```
	@id AS INTEGER
AS BEGIN
	SELECT first, last, district FROM person WHERE person_id = @id;
END;

GO
CREATE OR ALTER PROCEDURE find_local_candidates
	@voter_id AS INTEGER
AS BEGIN
	DECLARE @district VARCHAR(5);

	SELECT @district = district FROM person WHERE person_id = @voter_id;

	SELECT person_id, first, last, district, election.election_id, date as
election_date, campaign_id FROM campaign
	INNER JOIN person ON campaign.candidate_id = person_id
	INNER JOIN election ON campaign.election_id = election.election_id
	WHERE district = @district AND person_id != @voter_id
END;


GO
CREATE OR ALTER PROCEDURE query_events
	@candidate_id AS INTEGER = NULL,
	@type_filter AS VARCHAR(20)
AS BEGIN
	SET NOCOUNT ON;

	SELECT * FROM event
	LEFT JOIN campaign ON event.campaign_id = campaign.campaign_id
	LEFT JOIN person ON campaign.candidate_id = person.person_id
	WHERE (
	@candidate_id IS NULL OR person.person_id = @candidate_id
	) AND (
	@type_filter IS NULL
	OR EXISTS (
		SELECT value FROM STRING_SPLIT(@type_filter, ',')
		WHERE value = event.type
	)
	) AND (
	time_end > SYSUTCDATETIME()
	)
	ORDER BY time_start ASC
END;
```

```
GO
CREATE OR ALTER PROCEDURE register_for_event
       @event_id AS INTEGER,
       @person_id AS INTEGER,
       @utm AS VARCHAR(255)
AS BEGIN
       SET NOCOUNT ON;
       SET IMPLICIT_TRANSACTIONS OFF;
       BEGIN TRANSACTION;

       DECLARE @attendees INTEGER, @capacity INTEGER;

       SELECT @attendees = COUNT(*) FROM attend_event WHERE event_id =
@event_id;
       SELECT @capacity = max_capacity FROM event WHERE event_id = @event_id;

       IF @attendees + 1 > @capacity
       ROLLBACK TRANSACTION;
       ELSE BEGIN
       INSERT INTO attend_event(
              event_id,
              person_id,
              register_utm,
              register_timestamp
       ) VALUES (
              @event_id,
              @person_id,
              @utm,
              SYSUTCDATETIME()
       );
       COMMIT TRANSACTION;
       END;
END
GO
```

Java UI Code

First, there is a helper class Cli to manage UI stuff:

```
package CampaignManager;

import java.util.Arrays;
import java.util.Scanner;
```

```java
public class Cli {
    private Scanner scanner;
    public Cli(Scanner scanner) {
        this.scanner = scanner;
    }

    public Scanner getScanner() {
        return this.scanner;
    }

    public boolean askBool(String msg) {
        String cmd = "";
        while (!cmd.equals("y") && !cmd.equals("n")) {
            System.out.println(msg + " (y/n)");
            cmd = scanner.nextLine().trim().toLowerCase();
        }

        return cmd.equals("y");
    }

    public int askInt(String msg) {
        System.out.println(msg + " (enter an integer)");
        var i = scanner.nextInt();
        scanner.nextLine();
        return i;
    }

    public int choice(String question, String... choices) {
        int i = 0;
        for (String choice: choices) {
            System.out.println("  " + choice + " (" + (1 + i++) + ")");
        }

        while (true) {
            System.out.println(question + " (enter an integer 1 to " + i +
")");
            var ans = scanner.nextInt();
            scanner.nextLine();
            if (0 < ans && ans <= i) {
                return ans - 1;
```

```
            } else {
                System.out.println("Not in range.");
            }
        }
    }

    public int[] selectMultiple(String question, String... choices) {
        int i = 0;
        for (String choice: choices) {
            System.out.println("  " + choice + " (" + (1 + i++) + ")");
        }

        while (true) {
            System.out.println(question + " (enter a list of integers 1 to
" + i + ", e.g. 1,3,4 or \"none\" for none)");
            var ans = scanner.nextLine().trim();
            if (ans.equals("none")) {
                return new int[] {};
            }

            var arr = Arrays.stream(ans.split(",")).mapToInt(s ->
Integer.parseInt(s) - 1).toArray();
            if (Arrays.stream(arr).allMatch(k -> 0 <= k && k <
choices.length)) {
                return arr;
            }
        }
    }
}
```

And a function in Utility.java:

```
public static int login(Connection conn, Cli cli) throws Exception {
        var voter_id = cli.askInt("What is your user ID?");
        var call = conn.prepareCall("{call get_person(?)}");
        call.setInt(1, voter_id);
        var result = call.executeQuery();

        if (!result.next()) {
            throw new Exception("Person with id " + voter_id + " does not
exist.");
        }
```

```
        System.out.println(
            "You are logged in as "
            + result.getString("first") + " "
            + result.getString("last") + " residing in "
            + result.getString("district")
        );

        return voter_id;
    }
```

Now, for the use-case specific code:

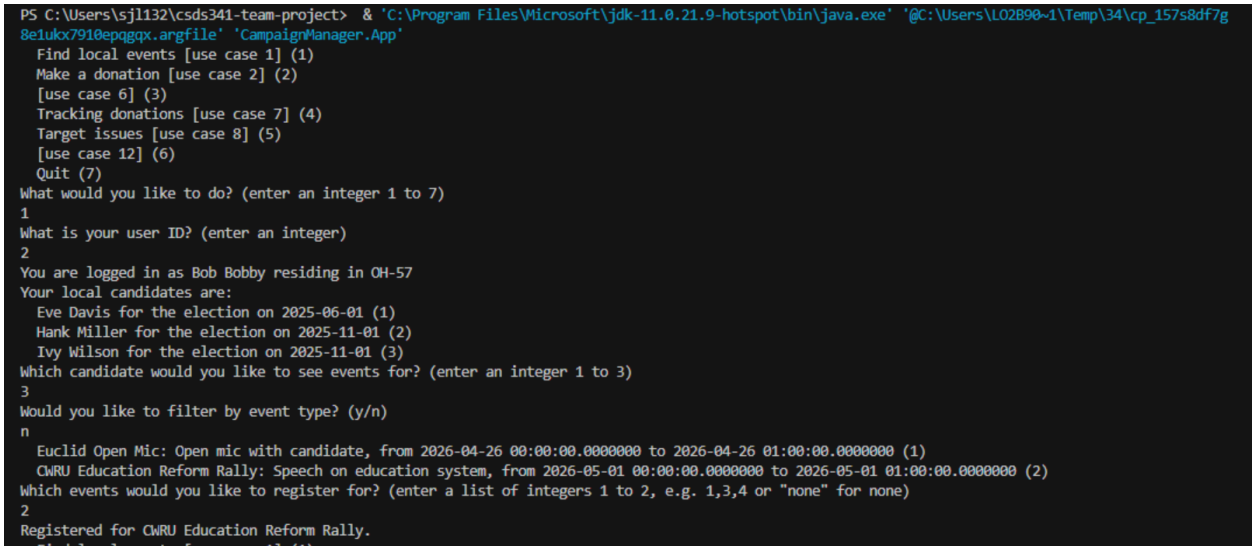
```
package CampaignManager;

import java.sql.Connection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Collectors;

public class UseCase1 {
    public static void run(Connection conn, Cli cli) throws Exception {
        var uid = Utility.login(conn, cli);

        var findLocalCall = conn.prepareCall("{ call
find_local_candidates(?) }");
        findLocalCall.setInt(1, uid);
        var findLocalResult = findLocalCall.executeQuery();

        var candidateIds = new ArrayList<>();
        var candidateChoices = new ArrayList<>();
        while (findLocalResult.next()) {
            candidateIds.add(findLocalResult.getInt("person_id"));

            candidateChoices.add(
                findLocalResult.getString("first") + " " +
                findLocalResult.getString("last") + " for the election on "
+
                findLocalResult.getString("election_date")
            );
        }
```

```
        System.out.println("Your local candidates are:");
        var id = cli.choice(
            "Which candidate would you like to see events for?",
            candidateChoices.toArray(String[]::new)
        );

        String[] types = {
            "town hall",
            "gotv",
            "rally",
            "phone bank",
            "other"
        };

        String filter = null;
        if (cli.askBool("Would you like to filter by event type?")) {
            filter = Arrays.stream(cli.selectMultiple("Select the events
you would like to include", types))
                .mapToObj(k -> types[k])
                .collect(Collectors.joining(","));
        }

        var queryEventsCall = conn.prepareCall("{ call query_events(?, ?)
}");
        queryEventsCall.setInt(1, id);
        queryEventsCall.setString(2, filter);

        var queryEventsResult = queryEventsCall.executeQuery();
        var eventIds = new ArrayList<Integer>();
        var eventChoices = new ArrayList<>();
        var eventNames = new ArrayList<>();
        while (queryEventsResult.next()) {
            eventIds.add(queryEventsResult.getInt("event_id"));

            eventChoices.add(
                queryEventsResult.getString("name") + ": " +
                queryEventsResult.getString("description") + ", from " +
                queryEventsResult.getString("time_start") + " to " +
                queryEventsResult.getString("time_end")
```

```
            );

            eventNames.add(queryEventsResult.getString("name"));
        }

        var events = cli.selectMultiple("Which events would you like to
register for?", eventChoices.toArray(String[]::new));
        for (int i: events) {
            var call = conn.prepareCall("{ call register_event(?, ?, ?)
}");
            call.setInt(1, eventIds.get(i));
            call.setInt(2, uid);
            call.setString(3, "utm_source=cli");

            call.execute();
            System.out.println("Registered for " + eventNames.get(i) +
".");
        }
    }
}
```

Screenshots

```
PS C:\Users\sjl132\csds341-team-project> & 'C:\Program Files\Microsoft\jdk-11.0.21.9-hotspot\bin\java.exe' '@C:\Users\LO2B90~1\Temp\34\cp_157s8df7g
8e1ukx7910epqgqx.argfile' 'CampaignManager.App'
  Find local events [use case 1] (1)
  Make a donation [use case 2] (2)
  [use case 6] (3)
  Tracking donations [use case 7] (4)
  Target issues [use case 8] (5)
  [use case 12] (6)
  Quit (7)
What would you like to do? (enter an integer 1 to 7)
1
What is your user ID? (enter an integer)
1
You are logged in as John James residing in OH-99
Your local candidates are:
  Frank Moore for the election on 2025-11-01 (1)
  Jack Taylor for the election on 2025-11-01 (2)
Which candidate would you like to see events for? (enter an integer 1 to 2)
2
Would you like to filter by event type? (y/n)
y
  town hall (1)
  gotv (2)
  rally (3)
  phone bank (4)
  other (5)
Select the events you would like to include (enter a list of integers 1 to 5, e.g. 1,3,4 or "none" for none)
1,2,3,4,5
  Northwood Kickoff: Kickoff rally in Northwood, from 2026-04-25 00:00:00.0000000 to 2026-04-25 01:00:00.0000000 (1)
  University Hospitals Town Hall: Focused on healthcare reform, from 2026-04-30 00:00:00.0000000 to 2026-04-30 01:00:00.0000000 (2)
Which events would you like to register for? (enter a list of integers 1 to 2, e.g. 1,3,4 or "none" for none)
2
```



2. Donors looking to make a donation (Serena Lynas)

Description: Politicians usually rely on donors and special interest groups to make contributions that finance their campaign. It is in the candidate's interest to make donating as seamless as possible and to ask for donations frequently. This application will allow donors, both large and small, to search through candidates and find those who best align with their interests, and make a contribution online. It will also check donations against the statutory limit to prevent donors from violating campaign finance laws.

Details: From the potential donor side, things are a little bit different, because we don't want to filter by local candidates. Frequently, donors will donate towards campaigns outside of their district; for example the recent Congressional special elections in Florida that drew in millions of dollars from all over the nation. Therefore, for donations, the returned results are oriented towards filtering for candidates that support specific issues so that special interest groups can identify candidates that they would like to donate towards. The system will first query for a list of issues (`get_issues`, which is just a simple SELECT statement – it is a small table), then the user will indicate which issue they are interested in, and then the system will use the stored procedure `find_candidates_by_issue` to show which candidates and campaigns support an issue. This procedure uses a SELECT with a join on the person, campaign, election, person_issue, and issue tables, and also filters for elections that are in the future (i.e. have not already occurred). Lastly, the user will be prompted for which campaign they would like to donate towards and for the amount, and it will check to ensure that it is within the statutory limit. The SQL code will also compute a sum over all of the past donations in this election cycle to ensure they will not go over, and I used a transaction here because again we need atomicity – we do not want the summation of the total donated in the election cycle to become incorrect if there are two concurrent donations from the same person (if they were being clever to avoid campaign finance laws). Lastly, the donation is inserted into the table (2 DML used – SELECT and INSERT). This also will cause a trigger to fire that keeps track of each campaign's total

funds – this is elaborated more in another group member's use case (they wrote the trigger, but it applies to both use cases).

User Requirements: The requirements are only that the donor know their User ID in order to be able to log in (so that their donations can be correctly attributed). Valid user IDs include the range of 1 through 10. Everything else will be prompted by the system, given that the list of candidates which match a certain issue isn't empty.

SQL Code:

```
CREATE OR ALTER PROCEDURE get_issues AS BEGIN
   SELECT * FROM issue
END;
GO

CREATE OR ALTER PROCEDURE find_candidates_by_issue
   @issue_id AS INTEGER
AS BEGIN
   SELECT person.person_id, first, last, district, campaign_id,
election.election_id, date as election_date
   FROM person
   INNER JOIN campaign ON campaign.candidate_id = person.person_id
   INNER JOIN election ON election.election_id = campaign.election_id
   INNER JOIN person_issue ON person_issue.person_id = person.person_id
   INNER JOIN issue ON person_issue.issue_id = issue.issue_id
   WHERE date > GETDATE()
END;
GO

CREATE OR ALTER PROCEDURE make_donation
   @person_id INTEGER,
   @campaign_id INTEGER,
   @amount BIGINT,
   @statutory_limit BIGINT
AS
BEGIN
   BEGIN TRANSACTION;
   -- Do not allow donations to campaigns in the past.
   DECLARE @date DATE, @election_id INTEGER;

   SELECT @date = date, @election_id = campaign.election_id FROM election
   INNER JOIN campaign ON campaign.election_id = election.election_id
```

```
    WHERE campaign.campaign_id = @campaign_id;

    IF @date IS NULL OR @date < GETDATE() BEGIN
        ROLLBACK TRANSACTION;
        THROW 51000, 'Cannot donate to an election in the past', 1;
    END ELSE BEGIN
        DECLARE @cycle_sum BIGINT;

        SELECT @cycle_sum = SUM(amount) FROM donation
        INNER JOIN campaign ON campaign.campaign_id = donation.campaign_id
        WHERE person_id = @person_id AND election_id = @election_id;

        IF @cycle_sum + @amount > @statutory_limit BEGIN
            ROLLBACK TRANSACTION;
            THROW 51001, 'Donation would exceed statutory limit', 2;
        END ELSE BEGIN
            INSERT INTO donation VALUES (@person_id, @campaign_id,
@amount);
            COMMIT TRANSACTION;
        END
    END
END;
GO
```

Java UI Code

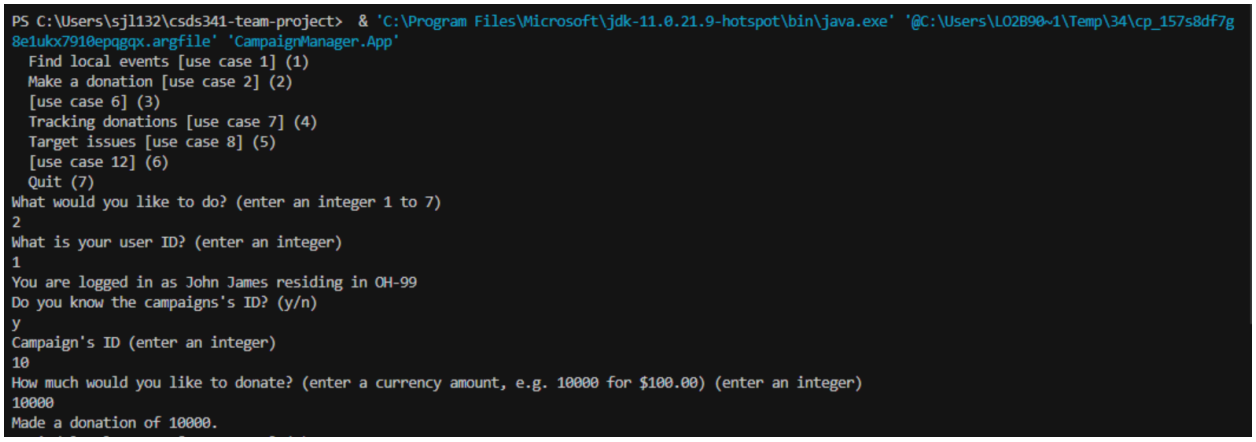
The same utility classes are used as the previous use case. For the use-case specific code:

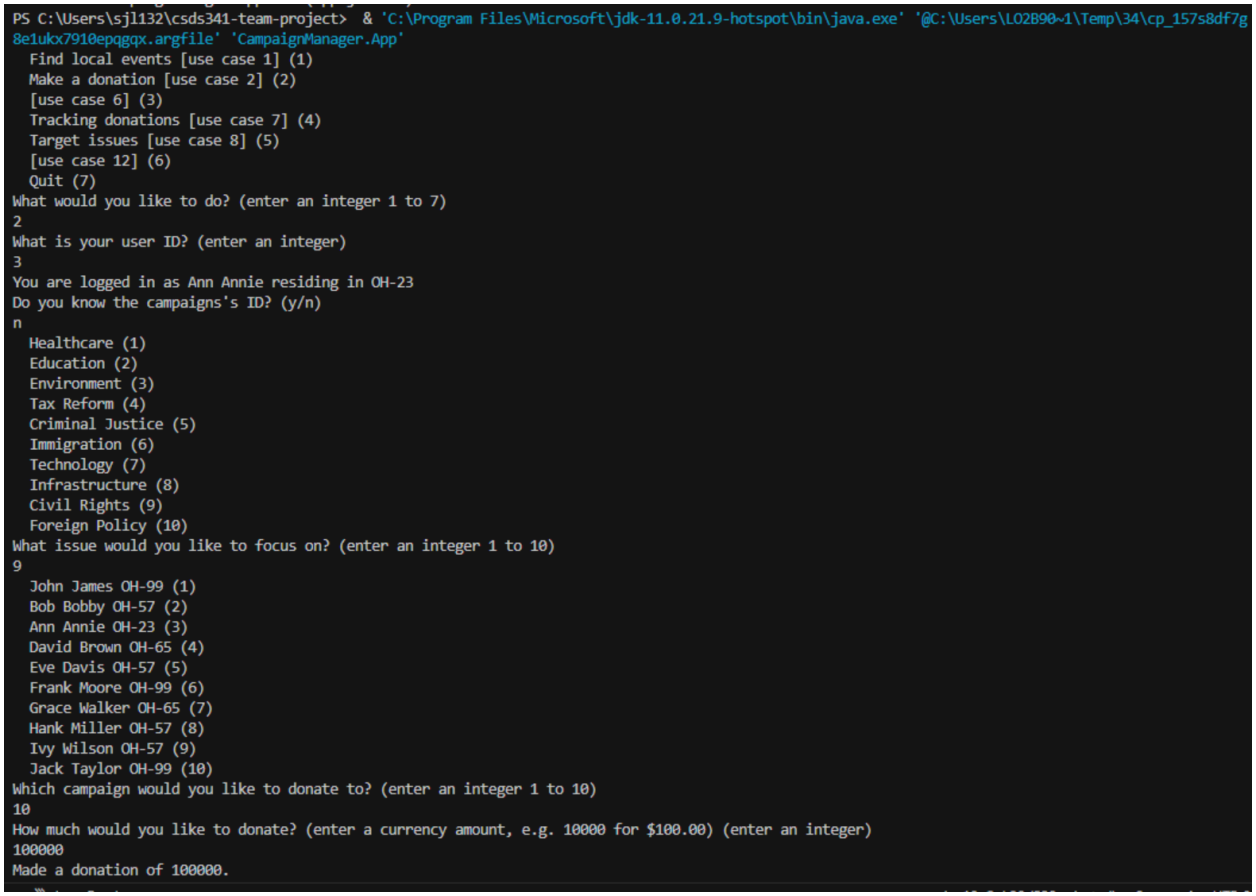
```
package CampaignManager;

import java.sql.Connection;
import java.util.ArrayList;

public class UseCase2 {
    public static int STATUTORY_LIMIT = 5500000;

    public static void run(Connection conn, Cli cli) throws Exception {
        var uid = Utility.login(conn, cli);

        var issuesResult = conn.prepareCall("{ call get_issues()
}").executeQuery();
```

```
        int campaign_id;
        if (cli.askBool("Do you know the campaigns's ID?")) {
            campaign_id = cli.askInt("Campaign's ID");
        } else {
            var issueIds = new ArrayList<Integer>();
            var issueDescs = new ArrayList<>();
            while (issuesResult.next()) {
                issueIds.add(issuesResult.getInt("issue_id"));
                issueDescs.add(issuesResult.getString("description"));
            }

            var issueIndex = cli.choice("What issue would you like to focus
on?", issueDescs.toArray(String[]::new));
            var issueId = issueIds.get(issueIndex);

            var campaignIds = new ArrayList<Integer>();
            var candidateDesc = new ArrayList<>();
            var findCandidatesCall = conn.prepareCall("{ call
find_candidates_by_issue(?) }");
            findCandidatesCall.setInt(1, issueId);
            var result = findCandidatesCall.executeQuery();

            while (result.next()) {
                campaignIds.add(result.getInt("campaign_id"));
                candidateDesc.add(
                    result.getString("first") + " " +
                    result.getString("last") + " " +
                    result.getString("district")
                );
            }

            campaign_id = campaignIds.get(
                cli.choice("Which campaign would you like to donate to?",
candidateDesc.toArray(String[]::new))
            );
        }

        int amnt = cli.askInt("How much would you like to donate? (enter a
currency amount, e.g. 10000 for $100.00)");
```

```
        if (amnt > STATUTORY_LIMIT) {
            System.out.println("This is above the statutory limit.");
            return;
        }

        var call = conn.prepareCall("{ call make_donation(?, ?, ?, ?) }");
        call.setInt(1, uid);
        call.setInt(2, campaign_id);
        call.setInt(3, amnt);
        call.setInt(4, STATUTORY_LIMIT);

        call.execute();

        System.out.println("Made a donation of " + amnt + ".");
    }
}
```

Screenshots





3. People looking to volunteer for their candidate (Serena Lynas – Not implemented)

During the course of a contentious campaign season, often members of the general public will want to volunteer their time, such as for phone banks, door knocking, etc. in order to help out their favored candidate. This application will allow aspiring volunteers to search for and register themselves to volunteer for a candidate.

Details: This use case is similar to the prior use case, but users may want to volunteer in elections outside of their district. Additionally, they will be more interested in get-out-to-vote campaigns and phone banks as an organizer rather than as a member of the general public. This use case would use a SELECT statement to find events connected to a candidate, and an INSERT statement on `attend_event` to sign the user up. A transaction can also be used if an operation may require a rollback if it doesn't work, or atomicity is required (as seen in the other use cases).

User requirements: This is intended for people who are already politically engaged. They must already know the ID of the candidate they are looking to volunteer for and their own ID. Everything else, like the events that they can attend, will be provided as the result of a query. There is no requirement for someone to volunteer; any user is able to volunteer, provided they know who they want to volunteer for.

Organizing

4. Searching for possible venues (Serena Lynas – Not Implemented)

It can be difficult to find venues to host political events. Organizers who want to search for a venue to host their event with a desired capacity can do so. An example is an organizer trying to find out what venue is best for a rally. This application will allow filtering of venues based on the most used venues by an event type and capacity, and the user to create a new event at a venue.

Details: This selects from the venues table, filtering optionally for location. It would have used a `SELECT` statement on the venues table to find the appropriate venues, and it would have joined on the event table to make sure no venue was double-booked. For booking, a transaction would have been used for atomicity to make sure that a simultaneous booking wouldn't cause the same venue to be used for different events. An `INSERT` statement on `event` would have been used to create the event at the venue. (Two different kinds of DML statements).

User requirements: desired minimum capacity, optionally location for filtering. Everything else will be prompted.

5. Searching for Volunteers (Angel Diaz - Not Implemented)

Description: In the event of staffing shortfalls, organizers need to reach out to possible volunteers to fill these gaps. People who have volunteered in the past are more likely to volunteer again, so campaign organizers should look to re-recruit volunteers from previous campaign seasons first to fill these shortfalls. The application will allow these organizers to identify previous volunteers based on which candidate and which elections they have previously volunteered for, and it will give them their contact information so that they may be reached again.

Details: This use case helps campaign organizers find people who have previously volunteered for a specific candidate or election. Volunteer activity is recorded in attend_event through specific event types such as "phone bank" or "gotv". Organizers can filter by candidate name, election ID, and event type. The application returns a distinct list of previous volunteers along with their contact information, and optionally logs the search results to a volunteer_contact_log table

User Requirements: candidate_name, election_id, and event_type (all optional, used for filtering)

6. Find New Candidates (Angel Diaz - Implemented)

Description: Some districts don't have anyone running, and districts that need candidates could benefit from this application. The application will query by election history in a district and the

desired issues wanted in a candidate to determine which person would be best to run. Additionally, a person's contact information can be obtained for them to be contacted.

Details: This requires information from the person, person_issue, issue, campaign, and election tables to be SELECTED, then candidates are filtered based on how many elections they've been in and if their issue matches the issue input. The user then gets displayed the top 5 candidates and can select one. They will then be INSERTED into the campaign table.

User Requirements: This use case requires a district, issueId, electionId, and managerId to function correctly. The CLI interface will provide available options if needed.

SQL Code:

```
Unset
DROP PROCEDURE IF EXISTS find_new_candidates;
GO
CREATE PROCEDURE find_new_candidates
    @district VARCHAR(5),
    @issueId INT,
    @electionId INT
AS
BEGIN
    SET NOCOUNT ON;

    CREATE TABLE #CandidateScores (
        person_id INT,
        first NVARCHAR(100),
        last NVARCHAR(100),
        email NVARCHAR(100),
        phone NVARCHAR(20),
        elections_participated INT,
        matching_issue INT
    );

    INSERT INTO #CandidateScores (person_id, first, last, email, phone,
elections_participated, matching_issue)
    SELECT
        p.person_id,
        p.first,
        p.last,
        p.email,
        p.phone,
        (SELECT COUNT(*) FROM campaign c2 WHERE c2.candidate_id = p.person_id)
AS elections_participated,
```

```
        (SELECT COUNT(*) FROM person_issue pi2 WHERE pi2.person_id =
p.person_id AND pi2.issue_id = @issueId) AS matching_issue
    FROM person p
    LEFT JOIN campaign c ON c.candidate_id = p.person_id AND c.election_id =
@electionId
    WHERE p.district = @district
      AND c.campaign_id IS NULL;


    SELECT TOP 5 *
    FROM #CandidateScores
    WHERE matching_issue > 0
    ORDER BY elections_participated DESC, matching_issue DESC;

    DROP TABLE #CandidateScores;
END
GO
```

Unset

```
DROP PROCEDURE IF EXISTS add_candidate_to_campaign;
GO
CREATE PROCEDURE add_candidate_to_campaign
    @personId INT,
    @electionId INT,
    @managerId INT
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;

        IF EXISTS (
            SELECT 1 FROM campaign
            WHERE candidate_id = @personId AND election_id = @electionId
        )
        BEGIN
            RAISERROR('Candidate is already registered for this election.', 16,
1);
            ROLLBACK;
            RETURN;
        END;

        INSERT INTO campaign (candidate_id, manager_id, election_id, funds)
```

```
        VALUES (@personId, @managerId, @electionId, 0.00);

        COMMIT;
    END TRY
    BEGIN CATCH
        ROLLBACK;
        PRINT 'Error: ' + ERROR_MESSAGE();
    END CATCH
END
GO
```

Java UI Code:

Unset

```
package CampaignManager;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.Scanner;

public class UseCase6 {
    public static void run(Connection conn, Scanner scanner) throws
SQLException {
        System.out.println("Use Case 6: Identify and add a new candidate to an
upcoming election.");


        display_districts(conn);
        System.out.println("Enter the district you would like to find a
candidate for:");
        String district = scanner.nextLine();


        String cmd = "";
        while (!cmd.equals("y") && !cmd.equals("n")) {
            System.out.println("Do you want to see a list of issues before
selecting (y/n)?");
            cmd = scanner.nextLine().toLowerCase();
        }
        if (cmd.equals("y")) {
            Utility.display_issues(conn);
        }
```

```
        System.out.println("Enter the issue_id that you want your candidate to
care about:");
        int issue_id = scanner.nextInt();
        scanner.nextLine();


        display_elections(conn);
        System.out.println("Enter the election_id to add a candidate for:");
        int election_id = scanner.nextInt();
        scanner.nextLine();


        display_campaign_managers(conn);
        System.out.println("Enter your person_id (as manager) from the above
list:");
        int manager_id = scanner.nextInt();
        scanner.nextLine();


        var callable = conn.prepareCall("{call find_new_candidates(?, ?, ?)}");
        callable.setString(1, district);
        callable.setInt(2, issue_id);
        callable.setInt(3, election_id);
        var result = callable.executeQuery();

        int index = 1;
        int[] person_ids = new int[5];

        System.out.println("\nTop recommended candidates:");

System.out.println("Index\tName\t\tEmail\t\tPhone\t\t#Elections\tMatching
Issue");
        while (result.next() && index <= 5) {
            int person_id = result.getInt("person_id");
            String first = result.getString("first");
            String last = result.getString("last");
            String email = result.getString("email");
            String phone = result.getString("phone");
            int elections = result.getInt("elections_participated");
            int matches = result.getInt("matching_issue");

            System.out.printf("%d\t%s %s\t%s\t%s\t%d\t\t%d\n", index, first,
last, email, phone, elections, matches);
```

```
            person_ids[index - 1] = person_id;
            index++;
        }

        if (index == 1) {
            System.out.println("no suitable candidates found for the district
and issue.");
            return;
        }


        System.out.println("Enter the index (1-5) of the person you'd like to
nominate as a candidate (or 0 to cancel):");
        int chosen_index = scanner.nextInt();
        scanner.nextLine();

        if (chosen_index < 1 || chosen_index > 5) {
            System.out.println("No candidate was added.");
            return;
        }

        int chosen_person_id = person_ids[chosen_index - 1];
        var insert_candidate = conn.prepareCall("{call
add_candidate_to_campaign(?, ?, ?)}");
        insert_candidate.setInt(1, chosen_person_id);
        insert_candidate.setInt(2, election_id);
        insert_candidate.setInt(3, manager_id);

        try {
            insert_candidate.executeUpdate();
            System.out.println("Candidate successfully added to the
campaign.");
        } catch (SQLException e) {
            System.out.println("Error while adding candidate: " +
e.getMessage());
        }
    }

    private static void display_districts(Connection conn) throws SQLException
{
        var stmt = conn.prepareStatement("SELECT DISTINCT district FROM person
WHERE district IS NOT NULL");
        var rs = stmt.executeQuery();
        System.out.println("\nAvailable Districts:");
```

```
        while (rs.next()) {
            System.out.println("- " + rs.getString("district"));
        }
        System.out.println();
    }

    private static void display_elections(Connection conn) throws SQLException
{
        var stmt = conn.prepareStatement(
            "SELECT election_id, date, registration_deadline " +
            "FROM election WHERE date >= CAST(GETDATE() AS DATE)"
        );
        var rs = stmt.executeQuery();
        System.out.println("\nUpcoming Elections:");
        while (rs.next()) {
            System.out.printf("election_id: %d | date: %s | registration
deadline: %s\n",
                rs.getInt("election_id"),
                rs.getDate("date"),
                rs.getDate("registration_deadline"));
        }
        System.out.println();
    }

    private static void display_campaign_managers(Connection conn) throws
SQLException {
        var stmt = conn.prepareStatement(
            "SELECT DISTINCT p.person_id, p.first, p.last, p.email " +
            "FROM person p " +
            "INNER JOIN campaign c ON p.person_id = c.manager_id"
        );
        var rs = stmt.executeQuery();
        System.out.println("\nExisting Campaign Managers:");
        while (rs.next()) {
            System.out.printf("person_id: %d | name: %s %s | email: %s\n",
                rs.getInt("person_id"),
                rs.getString("first"),
                rs.getString("last"),
                rs.getString("email"));
        }
        System.out.println();
    }}
```

Screenshots:

```
enter ucN to select the Nth use case, or q/Q to quit:
uc6
Use Case 6: Identify and add a new candidate to an upcoming election.

Available Districts:
- OH-57
- OH-65
- OH-99

Enter the district you would like to find a candidate for:
OH-99
Do you want to see a list of issues before selecting (y/n)?
y
issue_id: 1, description: Healthcare
issue_id: 2, description: Education
issue_id: 3, description: Environment
issue_id: 4, description: Tax Reform
issue_id: 5, description: Criminal Justice
issue_id: 6, description: Immigration
issue_id: 7, description: Technology
issue_id: 8, description: Infrastructure
issue_id: 9, description: Civil Rights
issue_id: 10, description: Foreign Policy
Enter the issue_id that you want your candidate to care about:
1

Upcoming Elections:
election_id: 1 | date: 2025-06-01 | registration deadline: 2025-05-01
election_id: 2 | date: 2025-11-01 | registration deadline: 2025-10-01

Enter the election_id to add a candidate for:
2

Existing Campaign Managers:
person_id: 11 | name: Karen Anderson | email: karen.anderson@example.com
person_id: 12 | name: Leo Thomas | email: leo.thomas@example.com
person_id: 13 | name: Mia Jackson | email: mia.jackson@example.com
person_id: 14 | name: Nick Harris | email: nick.harris@example.com
person_id: 15 | name: Olivia Martin | email: olivia.martin@example.com
person_id: 16 | name: Paul Lee | email: paul.lee@example.com
person_id: 17 | name: Quinn Walker | email: quinn.walker@example.com
person_id: 18 | name: Rachel Hall | email: rachel.hall@example.com
person_id: 19 | name: Sam Allen | email: sam.allen@example.com
person_id: 20 | name: Tina Young | email: tina.young@example.com

Enter your person_id (as manager) from the above list:
11

Top recommended candidates:
Index   Name            Email           Phone           #Elections      Matching Issue
1       Carol White     carol.white@example.com 555-3456        1               1
Enter the index (1-5) of the person you'd like to nominate as a candidate (or 0 to cancel):
1
Candidate successfully added to the campaign.
enter ucN to select the Nth use case, or q/Q to quit:
```

Data analytics & data science

7. Tracking donations (Ethan Kaji - Implemented)

It is desirable for campaign managers to track donations so that they can figure out which people are large donors so that they can be solicited again. This application should be able to handle identifying individuals who have historically made large donations and updating their recorded donations as they occur. It should be possible to get a list of contact information for prospective donors to help managers solicit donations. Finally, it should be possible to predict

which individuals may be likely to donate to a given campaign based on their previous donations.

Details:

This use case primarily works off of the `donation` relation. It provides functionality to `INSERT` and `DELETE` from the relation. It is possible that the person donating has not yet been entered into the database. In this case, the application should alert the user and query the necessary information to insert the new person. Additionally, the use case should support adding new campaigns, and possibly even elections, if they are not yet registered in the database. Both insertion and deletion are done within transactions through the Java frontend of our application.

Additionally, the funds within a campaign are directly dependent on the donations to that campaign. This use case includes two triggers that automatically update the total funds for a campaign as donations are added or removed. Another important part of this use case and use case eight is that they both allow users to look up or insert new people based on their first name, last name, and date of birth. We use an identity as the primary key inside our database, so we defined an index over the first name, last name, and date of birth in the person relation in order to make lookup of people more efficient.

In order to support identifying historically large donors, this use case also needs to be able to `SELECT` from an inner join between the `person` relation and the `donation` relation. This is how this use case determines high profile donors for any given campaign. The last thing needed is a selection over an inner join between `person_issue` and `campaign` which allows us to determine the issues relevant to a given campaign. From here, it is possible to compare two campaigns to identify the number of similar issues. If campaigns are similar enough, a campaign manager may decide to solicit donations from donors to another campaign.

This use case requires `INSERT`, `DELETE`, and `SELECT`, so there are three different DML statements in use.

User Requirements:

In order to use the implemented functions, the user must know `person_id` and `campaign_id` along with the donation `amount` for inserting a new donation. In the case that the donor does not exist in the database, the user must further provide the `first` and `last` name, date of birth (`dob`), phone number (`phone`) and email address (`email`), as well as the district the person is from (`district`). To delete a donation, the user only needs to know `donation_id`. To find the largest donors, the user needs to know just the `campaign_id`, and to find the similarities between two campaigns, only the `campaign_id` are needed.

SQL Code:

Unset

```
DROP INDEX IF EXISTS person_by_name_and_dob ON person;
CREATE INDEX person_by_name_and_dob
ON person(first, last, dob);
```

Unset

```
GO
CREATE OR ALTER TRIGGER add_funds_after_donation
ON donation
AFTER INSERT
AS
BEGIN
	UPDATE campaign
	SET funds = campaign.funds + donated.total_amount
	FROM campaign INNER JOIN (
	SELECT campaign_id, SUM(amount) as total_amount FROM inserted GROUP BY
campaign_id
	) AS donated ON campaign.campaign_id = donated.campaign_id;
END;
GO

GO
CREATE OR ALTER TRIGGER remove_funds_after_donation_removal
ON donation
AFTER DELETE
AS
BEGIN
	UPDATE campaign
	SET funds = campaign.funds - donated.total_amount
	FROM campaign INNER JOIN (
	SELECT campaign_id, SUM(amount) as total_amount FROM deleted GROUP BY
campaign_id
	) AS donated ON campaign.campaign_id = donated.campaign_id;
END;
GO
```

Unset

```
GO
CREATE OR ALTER PROCEDURE insert_person
```

```
	@first VARCHAR(255),
	@last VARCHAR(255),
	@dob DATE,
	@phone VARCHAR(255) = NULL,
	@email VARCHAR(255) = NULL,
	@address VARCHAR(255) = NULL,
	@district VARCHAR(255) = NULL
AS
BEGIN
	INSERT INTO person OUTPUT Inserted.person_id VALUES(@first, @last, @dob,
@phone, @email, @address, @district);
END;
GO
```

Unset

```
GO
CREATE OR ALTER PROCEDURE insert_donation
	@person_id INTEGER,
	@campaign_id INTEGER,
	@amount BIGINT
AS
BEGIN
	INSERT INTO donation VALUES (@person_id, @campaign_id, @amount);
END;
GO
```

Unset

```
GO
CREATE OR ALTER PROCEDURE delete_donation
	@donation_id INTEGER
AS
BEGIN
	DELETE FROM donation where donation.donation_id = @donation_id;
END;
GO
```

Unset

```
GO
CREATE OR ALTER PROCEDURE find_largest_donors
	@campaign_id INTEGER
AS
BEGIN
	SELECT person.first, person.last, person.phone, person.email,
total_donations
	FROM person INNER JOIN (
	SELECT donation.person_id, SUM(donation.amount) AS total_donations
	FROM donation
	WHERE donation.campaign_id = @campaign_id
	GROUP BY donation.person_id
	) AS donation_sum ON person.person_id = donation_sum.person_id
	ORDER BY donation_sum.total_donations;
END;
GO
```

Unset

```
-- We will use this to identify campaigns involving the same issues
GO
CREATE OR ALTER PROCEDURE check_campaign_similarity
	@campaign_a INTEGER,
	@campaign_b INTEGER
AS
BEGIN
	SELECT COUNT(*)
	FROM (
	SELECT person_issue.issue_id
	FROM person_issue INNER JOIN campaign ON person_issue.person_id =
campaign.candidate_id
	WHERE campaign.candidate_id = @campaign_a
	INTERSECT
	SELECT person_issue.issue_id
	FROM person_issue INNER JOIN campaign on person_issue.person_id =
campaign.candidate_id
	WHERE campaign.candidate_id = @campaign_b
	) as shared_issues;
END;
GO
```

Java Code:

Both use case 7 and use case 8 make substantial use of some shared utility codes for queries and insert:

Unset

```
package CampaignManager;

import java.sql.Connection;
import java.sql.Date;
import java.sql.SQLException;
import java.util.Scanner;

public class Utility {
	public static int login(Connection conn, Cli cli) throws Exception {
	var voter_id = cli.askInt("What is your user ID?");
	var call = conn.prepareCall("{call get_person(?)}");
	call.setInt(1, voter_id);
	var result = call.executeQuery();

	if (!result.next()) {
		throw new Exception("Person with id " + voter_id + " does not
exist.");
	}

	System.out.println(
		"You are logged in as "
		+ result.getString("first") + " "
		+ result.getString("last") + " residing in "
		+ result.getString("district")
	);

	return voter_id;
	}

	public static void display_issues(Connection conn) throws SQLException {
	var stmt = conn.prepareStatement("select * from issue");
	var result = stmt.executeQuery();
	while (result.next()) {
		int id = result.getInt("issue_id");
		String desc = result.getString("description");
		System.out.println("issue_id: " + id + ", description: " + desc);
	}
	}
```

```
	public static int fetch_or_insert_person(Connection conn, Scanner
scanner) throws SQLException {
	var fetch = conn
		.prepareStatement("select person_id from person where first = ?
and last = ? and dob = ?");
	var insert = conn.prepareCall("{call insert_person(?, ?, ?, ?, ?, ?,
?)}");

	System.out.println("first name:");
	var first = scanner.nextLine();

	System.out.println("last name:");
	var last = scanner.nextLine();

	System.out.println("date of birth (yyyy-[m]m-[d]d):");
	var dob = Date.valueOf(scanner.nextLine());

	fetch.setString(1, first);
	fetch.setString(2, last);
	fetch.setDate(3, dob);
	var fetch_result = fetch.executeQuery();

	while (fetch_result.next()) {
		var person_id = fetch_result.getInt("person_id");
		System.out.println("found someone with person_id = " + person_id);
		return fetch_result.getInt("person_id");
	}

	System.out.println("phone (empty for null):");
	var phone = scanner.nextLine();

	System.out.println("email (empty for null):");
	var email = scanner.nextLine();

	System.out.println("address (empty for null):");
	var address = scanner.nextLine();

	System.out.println("district (empty for null):");
	var district = scanner.nextLine();

	insert.setString(1, first);
	insert.setString(2, last);
	insert.setDate(3, dob);
	insert.setString(4, phone.length() > 0 ? phone : null);
```

```
        insert.setString(5, email.length() > 0 ? email : null);
        insert.setString(6, address.length() > 0 ? address : null);
        insert.setString(7, district.length() > 0 ? district : null);

        var insert_result = insert.executeQuery();
        insert_result.next();
        var person_id = insert_result.getInt("person_id");
        System.out.println("inserted someone with person_id = " + person_id);
        return person_id;
        }

        public static int fetch_or_insert_campaign(Connection conn, Scanner
scanner) throws SQLException {
        String cmd = "";
        while (!cmd.equals("y") && !cmd.equals("n")) {
                System.out.println("do you know the person_id of the campaign's
candidate (y/n/Y/N)?");
                cmd = scanner.nextLine().toLowerCase();
        }

        int candidate_id;
        if (cmd.equals("n")) {
                candidate_id = fetch_or_insert_person(conn, scanner);
        } else {
                System.out.println("enter the person_id:");
                candidate_id = scanner.nextInt();
                scanner.nextLine();
        }

        cmd = "";
        while (!cmd.equals("y") && !cmd.equals("n")) {
                System.out.println("do you know the person_id of the campaign's
manager (y/n/Y/N)?");
                cmd = scanner.nextLine().toLowerCase();
        }

        int manager_id;
        if (cmd.equals("n")) {
                manager_id = fetch_or_insert_person(conn, scanner);
        } else {
                System.out.println("enter the person_id:");
                manager_id = scanner.nextInt();
                scanner.nextLine();
        }
```

```
	cmd = "";
	while (!cmd.equals("y") && !cmd.equals("n")) {
		System.out.println("do you know the election_id of the campaign
(y/n/Y/N)?");
		cmd = scanner.nextLine().toLowerCase();
	}

	int election_id;
	if (cmd.equals("n")) {
		election_id = fetch_or_insert_election(conn, scanner);
	} else {
		System.out.println("enter the election_id:");
		election_id = scanner.nextInt();
		scanner.nextLine();
	}

	var select = conn.prepareStatement(
		"select campaign_id from campaign where candidate_id = ? and
manager_id = ? and election_id = ?");
	select.setInt(1, candidate_id);
	select.setInt(2, manager_id);
	select.setInt(3, election_id);
	var result = select.executeQuery();

	while (result.next()) {
		var campaign_id = result.getInt("campaign_id");
		System.out.println("found campaign with campaign_id = " +
campaign_id);
	}

	var insert = conn.prepareCall("{call insert_campaign(?, ?, ?)}");
	insert.setInt(1, candidate_id);
	insert.setInt(2, manager_id);
	insert.setInt(3, election_id);

	var insert_result = insert.executeQuery();
	insert_result.next();
	var campaign_id = insert_result.getInt("campaign_id");
	System.out.println("inserted a new campaign with campaign_id = " +
campaign_id);
	return campaign_id;
	}
```

```
	public static int fetch_or_insert_election(Connection conn, Scanner
scanner) throws SQLException {
	System.out.println("what is the date (yyyy-mm-dd) of the election?");
	var date = Date.valueOf(scanner.nextLine());

	System.out.println("what is the registration deadline (yyyy-mm-dd) of the
election?");
	var deadline = Date.valueOf(scanner.nextLine());

	var select = conn
		.prepareStatement("select election_id from election where date = ?
and registration_deadline = ?");
	select.setDate(1, date);
	select.setDate(2, deadline);

	var result = select.executeQuery();
	while (result.next()) {
		int election_id = result.getInt("election_id");
		System.out.println("found existing election with election_id = " +
election_id);
		return election_id;
	}

	var insert = conn.prepareCall("{call insert_election(?, ?)}");
	insert.setDate(1, date);
	insert.setDate(2, deadline);

	result = insert.executeQuery();
	result.next();

	int election_id = result.getInt("election_id");
	System.out.println("inserted new election with election_id = " +
election_id);
	return election_id;
	}

	public static void display_donations(Connection conn) throws SQLException
{
	var stmt = conn.prepareStatement("select * from donation");
	var result = stmt.executeQuery();

	while (result.next()) {
		System.out
```

```
                    .println("donation_id=" + result.getInt("donation_id") +
",person_id=" + result.getInt("person_id")
                            + ",campaign_id=" + result.getInt("campaign_id") +
",amount=" + result.getInt("amount"));
      }
      }

      public static void insert_donation(Connection conn, Scanner scanner)
throws SQLException {
      String cmd = "";
      while (!cmd.equals("y") && !cmd.equals("n")) {
            System.out.println("do you know the person_id of the donation's
donor (y/n/Y/N)?");
            cmd = scanner.nextLine().toLowerCase();
      }

      int donor_id;
      if (cmd.equals("n")) {
            donor_id = fetch_or_insert_person(conn, scanner);
      } else {
            System.out.println("enter the person_id of the donation's
donor:");
            donor_id = scanner.nextInt();
            scanner.nextLine();
      }

      cmd = "";
      while (!cmd.equals("y") && !cmd.equals("n")) {
            System.out.println("do you know the campaign_id of the campaign
the donation was made too (y/n/Y/N)?");
            cmd = scanner.nextLine().toLowerCase();
      }

      int campaign_id;
      if (cmd.equals("n")) {
            campaign_id = fetch_or_insert_campaign(conn, scanner);
      } else {
            System.out.println("enter the campaign_id:");
            campaign_id = scanner.nextInt();
            scanner.nextLine();
      }

      int donation_amount;
```

```
	System.out.println("enter the donation amount in cents (i.e. 1 dollar
becomes 100 cents):");
	donation_amount = scanner.nextInt();
	scanner.nextLine();

	var insert = conn.prepareCall("{call insert_donation(?, ?, ?)}");
	insert.setInt(1, donor_id);
	insert.setInt(2, campaign_id);
	insert.setInt(3, donation_amount);

	insert.executeUpdate();
	}

	public static void delete_donation(Connection conn, Scanner scanner)
throws SQLException {
	String cmd = "";
	while (!cmd.equals("n") && !cmd.equals("y")) {
		System.out.println("do you know the donation_id of the donation
you want to delete (y/n/Y/N)?");
		cmd = scanner.nextLine().toLowerCase();
	}

	if (cmd.equals("n")) {
		display_donations(conn);
	}

	System.out.println("enter the donation_id you want to delete:");
	var donation_id = scanner.nextInt();
	scanner.nextLine();

	var delete = conn.prepareCall("{call delete_donation(?)}");
	delete.setInt(1, donation_id);
	delete.executeUpdate();
	}
}
```

Unset

```
package CampaignManager;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.HashSet;
```

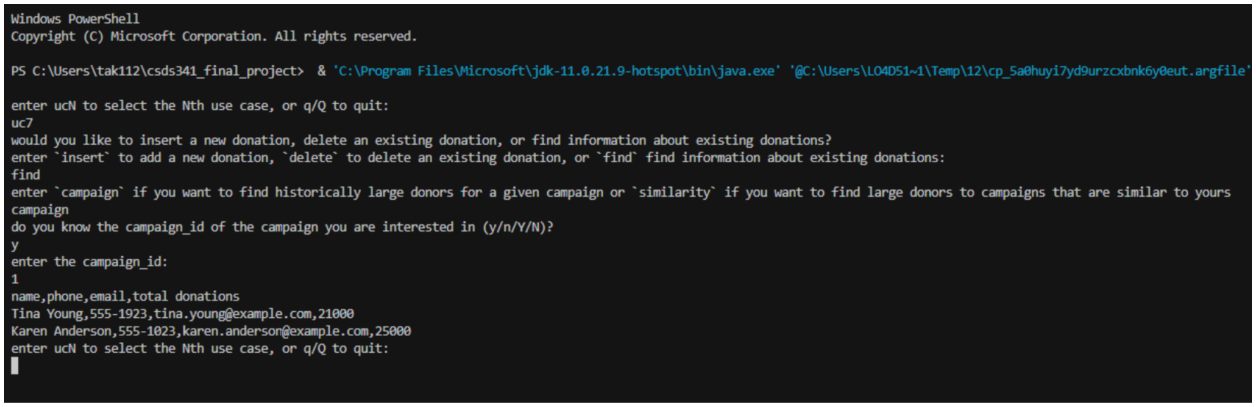
```
import java.util.PriorityQueue;
import java.util.Scanner;

public class UseCase7 {
	public static void run(Connection conn, Scanner scanner) throws
SQLException {
	System.out.println(
		"would you like to insert a new donation, delete an existing
donation, or find information about existing donations?");

	String cmd = "";
	while (!cmd.equals("insert") && !cmd.equals("find") &&
!cmd.equals("delete")) {
		System.out.println(
			"enter `insert` to add a new donation, `delete` to delete
an existing donation, or `find` find information about existing donations:");
		cmd = scanner.nextLine();
	}

	if (cmd.equals("insert")) {
		Utility.insert_donation(conn, scanner);
	} else if (cmd.equals("delete")) {
		Utility.delete_donation(conn, scanner);
	} else {
		find_donations(conn, scanner);
	}
	}

	private static void find_donations(Connection conn, Scanner scanner)
throws SQLException {

	String cmd = "";
	while (!cmd.equals("campaign") && !cmd.equals("similarity")) {
		System.out.println(
			"enter `campaign` if you want to find historically large
donors for a given campaign or `similarity` if you want to find large donors to
campaigns that are similar to yours");
		cmd = scanner.nextLine();
	}

	if (cmd.equals("campaign")) {
		find_donors_within_campaign(conn, scanner);
	} else {
		find_donors_across_campaigns(conn, scanner);
```

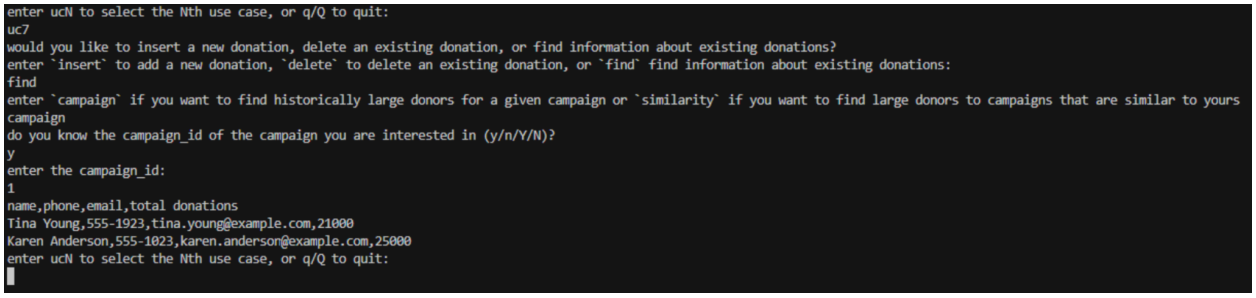
```
	}
	}

	private static void find_donors_within_campaign(Connection conn, Scanner
scanner) throws SQLException {
	var callable = conn.prepareCall("{call find_largest_donors(?)}");

	String cmd = "";
	while (!cmd.equals("y") && !cmd.equals("n")) {
		System.out.println("do you know the campaign_id of the campaign
you are interested in (y/n/Y/N)?");
		cmd = scanner.nextLine().toLowerCase();
	}

	int campaign_id;
	if (cmd.equals("n")) {
		campaign_id = Utility.fetch_or_insert_campaign(conn, scanner);
	} else {
		System.out.println("enter the campaign_id:");
		campaign_id = scanner.nextInt();
		scanner.nextLine();
	}

	callable.setInt(1, campaign_id);
	var result = callable.executeQuery();

	System.out.println("name,phone,email,total donations");
	while (result.next()) {
		System.out.println(result.getString("first") + " " +
result.getString("last") +
				"," + result.getString("phone") + "," +
result.getString("email") + "," +
				+ result.getInt("total_donations"));
	}
	}

	private static void find_donors_across_campaigns(Connection conn, Scanner
scanner) throws SQLException {
	System.out.println("this will find the top donors from the most similar
campaigns to yours.\n" +
		"How many campaigns would you like to generate donors from?");
	int count = scanner.nextInt();
	scanner.nextLine();
```

```
	var pq = new PriorityQueue<SimilarityResult>(count);

	String cmd = "";
	while (!cmd.equals("y") && !cmd.equals("n")) {
		System.out.println("do you know your campaign_id (y/n/Y/N)?");
		cmd = scanner.nextLine().toLowerCase();
	}

	int campaign_id;
	if (cmd.equals("n")) {
		campaign_id = Utility.fetch_or_insert_campaign(conn, scanner);
	} else {
		System.out.println("enter your campaign_id");
		campaign_id = scanner.nextInt();
		scanner.nextLine();
	}

	var fetch_all_campaign_ids = conn.prepareStatement("select campaign_id
from campaign where campaign_id != ?");
	fetch_all_campaign_ids.setInt(1, campaign_id);
	var campaign_similarity = conn.prepareCall("{call
check_campaign_similarity(?, ?)}");
	campaign_similarity.setInt(1, campaign_id);

	var other_campaign_ids = fetch_all_campaign_ids.executeQuery();

	// keep only the top count campaigns by number of similar issues
	while (other_campaign_ids.next()) {
		var similarity_result = new SimilarityResult();
		similarity_result.campaign_id =
other_campaign_ids.getInt("campaign_id");
		campaign_similarity.setInt(2, similarity_result.campaign_id);
		var similarity = campaign_similarity.executeQuery();
		similarity.next();
		similarity_result.count = similarity.getInt(1);

		pq.add(similarity_result);
		if (pq.size() > count) {
		pq.poll();
		}
	}

	var find_largest_donors = conn.prepareCall("{call
find_largest_donors(?)}");
```

```
	// for each campaign, collect the largest donors
	var top_donors = new HashSet<String>();
	while (!pq.isEmpty()) {
		var other_campaign_id = pq.poll().campaign_id;
		find_largest_donors.setInt(1, other_campaign_id);
		var result = find_largest_donors.executeQuery();

		while (result.next()) {
		top_donors.add(result.getString("first") + " " +
result.getString("last") + ","
				+ result.getString("phone") + "," +
result.getString("email"));
		}
	}

	System.out.println("name,phone,email");
	for (var donor : top_donors) {
		System.out.println(donor);
	}
	}

	static private class SimilarityResult implements
Comparable<SimilarityResult> {
	int count;
	int campaign_id;

	@Override
	public int compareTo(SimilarityResult other) {
		return Integer.compare(count, other.count);
	}
	}
}
```

Screenshots:





8. Tracking issues (Ethan Kaji - Implemented)

Since voters' issues of concern are tracked, campaign managers can use that information to generate issue-specific contact lists that they can email, text message, or mail that will be more effective than generic messaging. The applications should be able to generate a list of individuals who are likely to be concerned about issues at stake in a given election, and it should be able to produce the appropriate contact information in order to reach these individuals.

Details:

In order to implement this use case, the user needs to be able to `INSERT` into the `person_issue` table. It is possible that the person donating has not yet been entered into the database. In this case, the application should alert the user and query the necessary information to insert the new person. They also need to be able to select from an inner join between the `person_issue` table and the `person` table in order to identify all people who care deeply about a particular issue. Finally, they should be able to `SELECT` from the `campaign` table joined with the `person_issue` table. This can be used to produce a table of elections which were relevant to key issues based on the issues of the candidates. From here, an additional selection can be made in order to identify the people who voted in these elections, which can generate a list of people who are likely to be concerned about a given issue, even if they haven't explicitly stated they are.

This use case requires `INSERT` and `SELECT`, so there are two DML statements in use.

User Requirements:

In order to insert, the user only needs to know a `person_id` and an `issue_id`. In the case that the person does not exist in the database, the user must further provide the `first` and `last` name, date of birth (`dob`), phone number (`phone`) and email address (`email`), as well as the district the person is from (`district`). To identify people who care deeply about an issue, a user only needs an `issue_id`. To identify all elections associated with key issues, they again only need `issue_id`. Since this can be used as an intermediary table, the user again only needs `issue_id` in order to identify voters who have voted in an election with a given key issue at stake.

SQL Code:

```
Unset
GO
CREATE OR ALTER PROCEDURE insert_campaign
	@candidate_id INTEGER,
	@manager_id INTEGER,
	@election_id INTEGER
AS
BEGIN
	INSERT INTO campaign OUTPUT Inserted.campaign_id VALUES (@candidate_id,
@manager_id, @election_id, 0);
END;
GO
```

```
Unset
GO
CREATE OR ALTER PROCEDURE insert_election
	@date DATE,
	@registration_deadline DATE
AS
BEGIN
	INSERT INTO election OUTPUT Inserted.election_id VALUES
(@registration_deadline, @date);
END;
GO
```

```
Unset
GO
CREATE OR ALTER PROCEDURE find_people_for_issue
       @issue_id INTEGER
AS
BEGIN
       SELECT person.first, person.last, person.phone, person.email
       FROM person INNER JOIN person_issue ON person.person_id =
person_issue.person_id
       WHERE person_issue.issue_id = @issue_id;
END;
GO
```

```
Unset
GO
CREATE OR ALTER PROCEDURE find_elections_for_issue
       @issue_id INTEGER
AS
BEGIN
       SELECT DISTINCT campaign.election_id
       FROM campaign INNER JOIN person_issue ON campaign.candidate_id =
person_issue.person_id
       WHERE person_issue.issue_id = issue_id;
END;
GO
```

```
Unset
GO
CREATE OR ALTER PROCEDURE insert_person_issue
       @person_id INTEGER,
       @issue_id INTEGER
AS
BEGIN
       INSERT INTO person_issue OUTPUT Inserted.person_id, Inserted.issue_id
VALUES (@person_id, @issue_id);
END;
GO
```

```
Unset
GO
CREATE OR ALTER PROCEDURE voted_in
	@election_id INTEGER
AS
BEGIN
	SELECT person.first, person.last, person.phone, person.email
	FROM person INNER JOIN vote_cast ON person.person_id =
vote_cast.person_id
	WHERE vote_cast.election_id = @election_id
END;
GO
```

Java Code:

In addition to the utility code shared with use case 7, use case 8 makes use of the following java code:

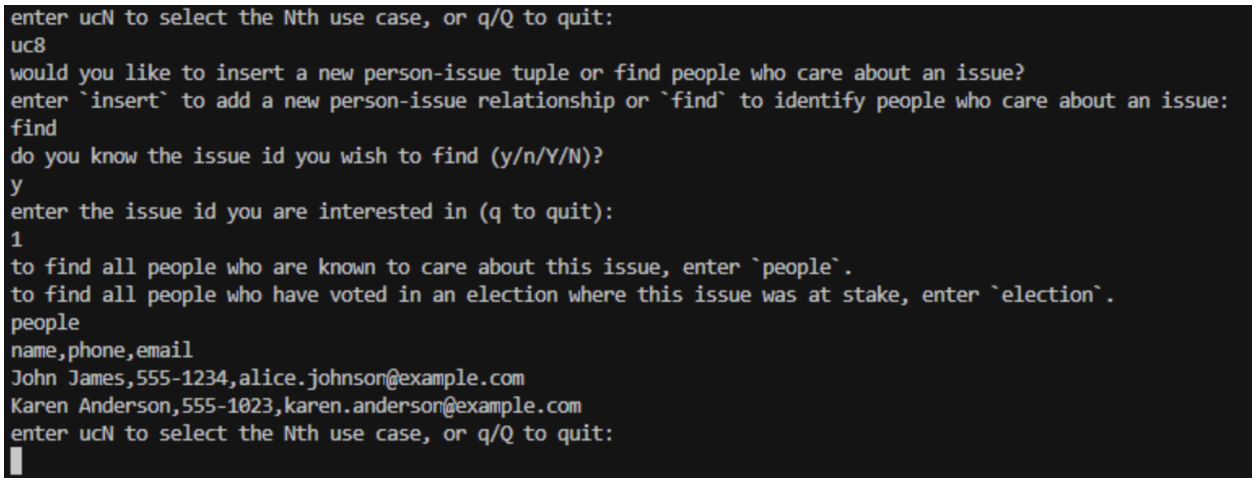
```
Unset
package CampaignManager;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.HashSet;
import java.util.Scanner;

public class UseCase8 {
	public static void run(Connection conn, Scanner scanner) throws
SQLException {
	System.out.println("would you like to insert a new person-issue tuple or
find people who care about an issue?");

	String cmd = "";
	while (!cmd.equals("insert") && !cmd.equals("find")) {
		System.out.println(
			"enter `insert` to add a new person-issue relationship or
`find` to identify people who care about an issue:");
		cmd = scanner.nextLine();
	}

	if (cmd.equals("insert")) {
		run_insert(conn, scanner);
```

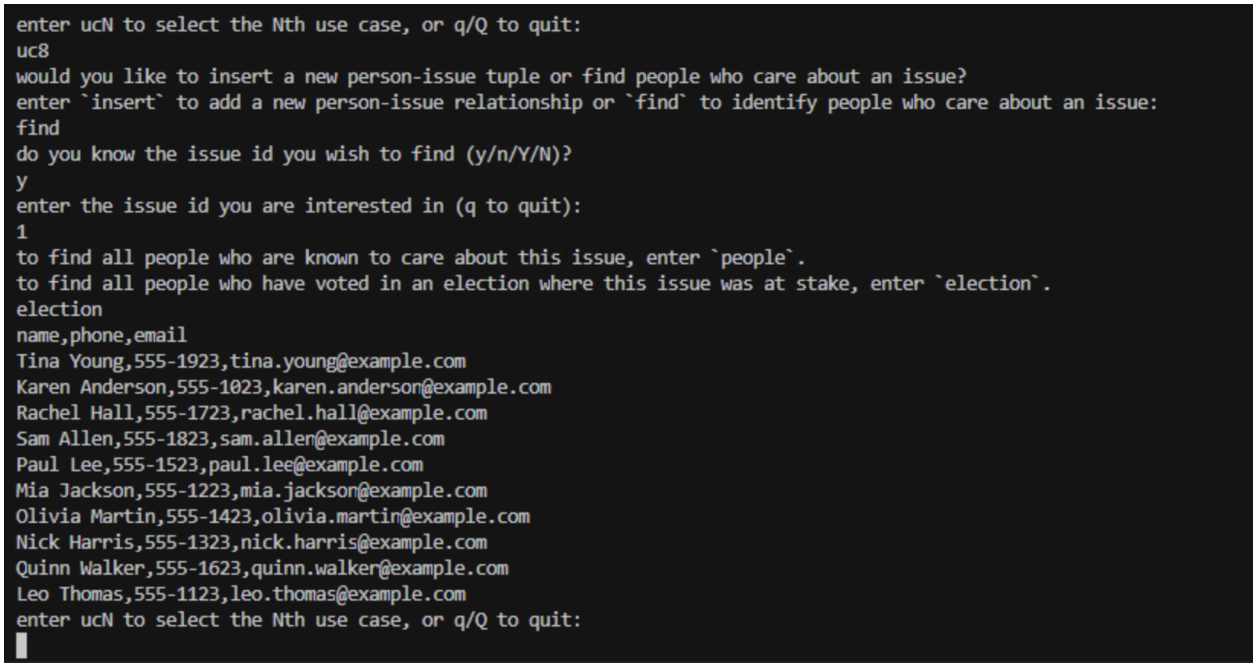
```
	} else {
		run_find(conn, scanner);
	}
	}

	private static void run_insert(Connection conn, Scanner scanner) throws
SQLException {
	String cmd = "";
	while (!cmd.equals("y") && !cmd.equals("n")) {
		System.out.println("do you know the person_id of the person you
want to edit (y/n/Y/N)?");
		cmd = scanner.nextLine().toLowerCase();
	}

	int person_id;
	if (cmd.equals("n")) {
		person_id = Utility.fetch_or_insert_person(conn, scanner);
	} else {
		System.out.println("enter the person_id:");
		person_id = scanner.nextInt();
		scanner.nextLine();
	}

	cmd = "";
	while (!cmd.equals("y") && !cmd.equals("n")) {
		System.out.println("do you know the issue_id of the issue you want
to connect this person to (y/n/Y/N)?");
		cmd = scanner.nextLine().toLowerCase();
	}

	if (cmd.equals("n")) {
		Utility.display_issues(conn);
	}

	System.out.println("enter the issue_id:");
	int issue_id = scanner.nextInt();
	scanner.nextLine();

	var insert = conn.prepareStatement("insert into person_issue values (?,
?)");
	insert.setInt(1, person_id);
	insert.setInt(2, issue_id);
	insert.executeUpdate();
	}
```

```
	private static void run_find(Connection conn, Scanner scanner) throws
SQLException {
	String cmd = "";
	while (!cmd.equals("y") && !cmd.equals("n")) {
		System.out.println("do you know the issue id you wish to find
(y/n/Y/N)?");
		cmd = scanner.nextLine().toLowerCase();
	}

	if (cmd.equals("n")) {
		Utility.display_issues(conn);
	}

	System.out.println("enter the issue id you are interested in (q to
quit):");
	var id_or_quit = scanner.nextLine();

	if (id_or_quit.toLowerCase().equals("q")) {
		scanner.close();
		return;
	}

	var issue_id = Integer.parseInt(id_or_quit);

	String where = "";
	while (!where.equals("people") && !where.equals("election")) {
		System.out.println("to find all people who are known to care about
this issue, enter `people`.");
		System.out.println(
			"to find all people who have voted in an election where
this issue was at stake, enter `election`.");
		where = scanner.nextLine();
	}

	if (where.equals("people")) {
		find_known_people(conn, issue_id);
	} else {
		find_by_vote(conn, issue_id);
	}
	}

	private static void find_known_people(Connection conn, int issue_id)
throws SQLException {
```

```
	var callable = conn.prepareCall("{call find_people_for_issue(?)}");
	callable.setInt(1, issue_id);
	var result = callable.executeQuery();

	System.out.println("name,phone,email");
	while (result.next()) {
		System.out.println(result.getString("first") + " " +
result.getString("last") + ","
			+ result.getString("phone") + "," +
result.getString("email"));
	}
	}

	private static void find_by_vote(Connection conn, int issue_id) throws
SQLException {
	var callable = conn.prepareCall("{call find_elections_for_issue(?)}");
	var voted_in = conn.prepareCall("{call voted_in(?)}");
	callable.setInt(1, issue_id);
	var result = callable.executeQuery();
	var lines = new HashSet<String>();
	while (result.next()) {
		var election_id = result.getInt("election_id");
		voted_in.setInt(1, election_id);
		var people = voted_in.executeQuery();

		while (people.next()) {
		lines.add(people.getString("first") + " " +
people.getString("last") + ","
			+ people.getString("phone") + "," +
people.getString("email"));
		}
	}

	System.out.println("name,phone,email");
	for (var line : lines) {
		System.out.println(line);
	}
	}
}
```

Screenshots:





9. Turnout prediction (Ethan Kaji - Not Implemented)

Using the tracked historical `vote_cast`, campaign managers can predict which voters are likely to turn out and which voters may need more encouragement and messaging from the campaign. The application should be able to help managers identify voters who have consistently voted in favor of specific candidates or issues. It should also be able to identify voters who likely care about specific issues, but may have not voted historically.

Details:

This use case requires users to be able to `INSERT` into the `vote_cast` relation. Note that it is often the case where new people may be added to the database only after they have voted once, so it should also be possible to `INSERT` a new person into the database if necessary

before adding a tuple into the `vote_cast` relation. The application needs to be able `SELECT` voters, based on the number of elections they have voted in historically. This can be done by selecting from the `vote_cast` table with an aggregation. The application should also be able to select voters based on the issues at stake in the elections they have voted on historically, so it should be able to select from a join between the `vote_cast` table and a set of elections where candidates cared about specific issues. This other table can be further generated using a join between the `campaign` table and the `person_issue` table.

This use case requires both `INSERT` and `SELECT`, so there are two DML statements required.

User Requirements:

In order to insert into the `vote_cast` relation, the user likely only needs to know the `person_id`, `election_id`, the `type` of the vote, and possibly who the person voted for. Note the time stamp of the vote is not necessary since this can be determined from when the person was entered into the database. If this is the first time a person is being entered into the database, they instead need to know the full identifying information of the person. This would require, `first` and `last` name, date of birth (`dob`), phone number (`phone`) and email address (`email`), as well as the district the person is from (`district`).

10. Outreach effectiveness (Ethan Kaji - Not Implemented)

Using the `utm` tracking data, the campaign can analyze which types of outreach and messaging are more effective in eliciting event attendance; and the campaign can use that data to optimize its messaging and driving event turnout. This application should be able to identify the various ways individuals who attended an event registered for said event. It should be able to generate aggregate statistics over said data allowing managers to change their advertising methods in order to improve attendance at future events. The scope of said aggregate information should be adjustable in order to allow managers to adjust their advertising based on their targets.

Details:

This would first require allowing users to `INSERT` people who registered for a given event. Since it is often the case where new people may be added to the database only after they have voted once, it should also be possible to `INSERT` a new person into the database if necessary. It is also possible that the event has not yet been registered in the database. In this case, the user should be queried for the necessary information to create a new event as well. Once the necessary events and people have been registered, the application needs to be able to query who attended events based on the `utm`. It should additionally be able to generate aggregate statistics over fields like `venue` as well in order to help users generate the best possible data.

This use case requires both `INSERT` and `SELECT`, so there are two DML statements required.

User Requirements:

In order to insert a new tuple into `attend_event`, the user only needs to know the `event_id`, `person_id`, and the `register_utm`. This is because the registration timestamp can be filled in using the database. Additionally, if a given person is not already registered in the database, the user must provide the `first` and `last` name, date of birth (`dob`), phone number (`phone`) and email address (`email`), as well as the district the person is from (`district`). The select requirements for aggregate statistics require selecting over a join between the `attend_event` relation and the `event_relation`, generating aggregates by grouping on different fields.

11. Voter Tracking (Angel Diaz - Not Implemented)

Description: Using voter addresses from the `person` relation as a way for campaign managers to see what key areas need to be targeted to gain more votes. This application will show managers districts and the most popular issues there. They can further filter to find out how their campaign issues are doing in certain areas, with the ones with the lowest support at the top.

Details: This use case lets campaign managers analyze how well their campaign platform aligns with voter interests across different districts. It SELECTS from the person, person_issue, issue, and campaign tables. Managers select a campaign and can optionally filter by district. The system returns a ranked list of issues by voter interest in that district, with the campaign's supported issues highlighted, and the ones with the least alignment shown at the top. This helps identify weak zones for targeted messaging. Optionally, results could be logged in a district_issue_support table for analysis over time.

User Requirements: This case would require the campaign_id to specify what interests and voters they should target. (optional: district)

12. Determine Event Effectiveness (Angel Diaz - Implemented)

Description: Events can be looked at in retrospect to determine what types of events have good turnout and the interests of the attendees can be correlated to determine which events resonate with which groups. This application will be able to collect aggregate statistics on various events as they occur in order to identify successful event types and venues.

Details: This use case analyzes event turnout and attendee interests. It SELECTS from the event, attend_event, person, and person_issue tables to determine which event types have the highest attendance and what issues attendees there care about most, inserting it into a temp table TopEvents. Then it inserts into EventSummary, which formats out the top 5 events with the issues and attendance. Users can filter by event type or campaign, and event names, attendance counts, and top attendee issues will be displayed.

User Requirements: This use case does not require the user to know anything beforehand, unless they are filtering for specific event types or campaigns. Event_Type and Campaign_Id

SQL Code:

```
Unset
DROP PROCEDURE IF EXISTS analyze_event_effectiveness;
GO
CREATE OR ALTER PROCEDURE analyze_event_effectiveness
    @eventType VARCHAR(50) = NULL,
    @campaignId INT = NULL
AS BEGIN
       CREATE TABLE #EventSummary(
    event_id INT,
    total_attendance INT,
    top_issue_id INT,
    top_issue_count INT,
       )

    SELECT TOP 5
        e.event_id,
        e.name AS event_name,
        COUNT(ae.person_id) AS total_attendance
    INTO #TopEvents
    FROM event e
    JOIN attend_event ae ON e.event_id = ae.event_id
    WHERE (@eventType IS NULL OR e.type = @eventType)
      AND (@campaignId IS NULL OR e.campaign_id = @campaignId)
    GROUP BY e.event_id, e.name
    ORDER BY total_attendance DESC;

    INSERT INTO #EventSummary(event_id, total_attendance, top_issue_id,
top_issue_count)
    SELECT te.event_id,
        te.total_attendance,
        pi.issue_id,
        COUNT(*) AS top_issue_count
    FROM #TopEvents te
    JOIN attend_event ae ON ae.event_id = te.event_id
    JOIN person_issue pi ON pi.person_id = ae.person_id
    GROUP BY te.event_id, te.total_attendance, pi.issue_id
    HAVING COUNT(*) = (
        SELECT MAX(issue_count) FROM (
            SELECT COUNT(*) AS issue_count
            FROM attend_event ae2
            JOIN person_issue pi2 ON pi2.person_id = ae2.person_id
            WHERE ae2.event_id = te.event_id
```

```
            GROUP BY pi2.issue_id
        ) AS counts
    );

    DROP TABLE #TopEvents;

       SELECT es.event_id, e.name, es.total_attendance, i.description AS
top_issue
       FROM #EventSummary es
       JOIN event e ON es.event_id = e.event_id
       JOIN issue i ON es.top_issue_id = i.issue_id;

       DROP TABLE #EventSummary;
END
GO
```

Java UI Code:

Unset

```
package CampaignManager;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.Scanner;

public class UseCase12 {
    public static void run(Connection conn, Scanner scanner) throws
SQLException {
        System.out.println("Use Case 12: Determine the effectiveness of past
events.");

        String event_type = null;
        String cmd = "";
        while (!cmd.equals("y") && !cmd.equals("n")) {
            System.out.println("Would you like to filter by event type
(y/n)?");
            cmd = scanner.nextLine().trim().toLowerCase();
        }
        if (cmd.equals("y")) {
            display_event_types(conn);
            System.out.println("Enter the event type to filter by:");
            event_type = scanner.nextLine();
```

```
        }

        Integer campaign_id = null;
        cmd = "";
        while (!cmd.equals("y") && !cmd.equals("n")) {
            System.out.println("Would you like to filter by campaign (y/n)?");
            cmd = scanner.nextLine().trim().toLowerCase();
        }
        if (cmd.equals("y")) {
            display_campaigns(conn);
            System.out.println("Enter the campaign_id to filter by:");
            campaign_id = scanner.nextInt();
            scanner.nextLine();
        }

        var callable = conn.prepareCall("{call analyze_event_effectiveness(?,
?)}");
        if (event_type != null && !event_type.isBlank()) {
            callable.setString(1, event_type);
        } else {
            callable.setNull(1, java.sql.Types.VARCHAR);
        }

        if (campaign_id != null) {
            callable.setInt(2, campaign_id);
        } else {
            callable.setNull(2, java.sql.Types.INTEGER);
        }

        var result = callable.executeQuery();

        System.out.println("\nTop 5 Events Based on Attendance and Attendee
Issues:");
        System.out.println("Event ID | Event Name\t\t| Attendance | Top
Issue");
        while (result.next()) {
            int event_id = result.getInt("event_id");
            String name = result.getString("name");
            int attendance = result.getInt("total_attendance");
            String top_issue = result.getString("top_issue");

            System.out.printf("%-9d| %-20s| %-11d| %s\n", event_id, name,
attendance, top_issue);
        }
```

```
    }

    private static void display_event_types(Connection conn) throws
SQLException {
        var stmt = conn.prepareStatement(
            "SELECT DISTINCT type FROM event WHERE type IS NOT NULL"
        );
        var rs = stmt.executeQuery();

        System.out.println("\nAvailable Event Types:");
        while (rs.next()) {
            System.out.println("- " + rs.getString("type"));
        }
        System.out.println();
    }

    private static void display_campaigns(Connection conn) throws SQLException
{
        var stmt = conn.prepareStatement(
            "SELECT c.campaign_id, p.first, p.last, e.date " +
            "FROM campaign c " +
            "JOIN person p ON c.candidate_id = p.person_id " +
            "JOIN election e ON c.election_id = e.election_id"
        );
        var rs = stmt.executeQuery();
        System.out.println("\nAvailable Campaigns:");
        while (rs.next()) {
            System.out.printf("campaign_id: %d | candidate: %s %s | election
date: %s\n",
                rs.getInt("campaign_id"),
                rs.getString("first"),
                rs.getString("last"),
                rs.getDate("date"));
        }
        System.out.println();
    }
}
```

Screenshots:

```
enter ucN to select the Nth use case, or q/Q to quit:
uc12
Use Case 12: Determine the effectiveness of past events.
Would you like to filter by event type (y/n)?
y

Available Event Types:
- gotv
- other
- phone bank
- rally
- town hall

Enter the event type to filter by (e.g. 'Town Hall', 'Debate'):
rally
Would you like to filter by campaign (y/n)?
n

Top 5 Events Based on Attendance and Attendee Issues:
Event ID | Event Name           | Attendance | Top Issue
1        | Northwood Kickoff    | 2          | Healthcare
1        | Northwood Kickoff    | 2          | Education
7        | Education Reform Rally| 2          | Technology
7        | Education Reform Rally| 2          | Infrastructure
enter ucN to select the Nth use case, or q/Q to quit:
uc12
Use Case 12: Determine the effectiveness of past events.
Would you like to filter by event type (y/n)?
n
Would you like to filter by campaign (y/n)?
y

Available Campaigns:
campaign_id: 1 | candidate: Alice Johnson | election date: 2025-06-01
campaign_id: 2 | candidate: Bob Smith | election date: 2025-06-01
campaign_id: 3 | candidate: Carol White | election date: 2025-06-01
campaign_id: 4 | candidate: David Brown | election date: 2025-06-01
campaign_id: 5 | candidate: Eve Davis | election date: 2025-06-01
campaign_id: 6 | candidate: Alice Johnson | election date: 2025-11-01
campaign_id: 7 | candidate: Grace Walker | election date: 2025-11-01
campaign_id: 8 | candidate: Hank Miller | election date: 2025-11-01
campaign_id: 9 | candidate: Ivy Wilson | election date: 2025-11-01
campaign_id: 10 | candidate: Jack Taylor | election date: 2025-11-01
campaign_id: 13 | candidate: Bob Smith | election date: 2025-11-01
campaign_id: 14 | candidate: Carol White | election date: 2025-11-01

Enter the campaign_id to filter by:
1

Top 5 Events Based on Attendance and Attendee Issues:
Event ID | Event Name           | Attendance | Top Issue
1        | Northwood Kickoff    | 2          | Healthcare
1        | Northwood Kickoff    | 2          | Education
6        | Healthcare Town Hall| 2          | Immigration
6        | Healthcare Town Hall| 2          | Technology
```

User Manual

Our CLI starts by providing a list of use cases for our user to select from. The parenthesis to the right of each option indicate that option's number; just type `1` for the option labeled `(1)` for example.

Each use case is provided with its associated title. By selecting a use case, the user is brought through a series of prompts which ask the user for the information necessary to complete the use case. This may involve requesting ID values for various people or attributes needed to look up the people within the database. Because this is a CLI application, it is important to get the input format correct. Usually, entering simple numbers or words is sufficient as detailed by the

prompts. In some cases, like with dates, the prompts request a specific format (yyyy-mm-dd), which is the format expected by the database application in order to properly parse a string into a SQL date. Since selecting a use case starts the operations needed to complete one, any failure to complete, either due to an exception or something else results in a commit rollback. This protects the database from corruption due to poor inputs. Additionally, after a use case has been completed, the application returns to the main menu, allowing the user to select another use case to implement.

Reflection

If we were able to start working on this project from the beginning, one of the main things we would change would be giving ourselves more time to work on deliverables. We ended up running into time crunches and just completing everything in one session for many of the steps while building this project. While it did not affect the quality of our work, it made it more stressful than it needed to be. Other than that we made efficient use of our time and given the chance to do things differently, we would set up meetings a week or two in advance to adequately give ourselves time for the project.

Another change we would make if we were given the change to redo this project would be to design the database around the use cases. One of the first things we did when building this project was design the relations we thought would be important for a campaign manager. However, we found that later, when implementing use cases, there were some parts of our database design that were detrimental to the use cases we wanted to implement, forcing use to use more convoluted SQL and Java code than necessary. If given the chance to start over, we would start by identifying the use cases we wanted to make for our application, then constructing the relations based on those use cases.

One of the biggest lessons we learned was making sure to check in with our TA whenever we were confused on what was needed for deliverables. It prevented us from potentially wasting time on developing non-essential project items. We also learned a lot about designing and implementing clean SQL and Java code, as well as many of the requirements for working on larger projects in groups.

Probably the greatest hurdle we had was initially figuring out where we could collaborate on code. There wasn't much direction on a recommended method and it took a couple different methods to find the best solution to use when collaborating on the project code. Initially we tried to live share code through Visual Studio Code but it was desyncing and causing issues. This resulted in us setting up a git and then setting up our individual environments in the virtual machine. From there we tested our SQL DDL, DML, and Java CLI, committing and merging pieces at a time. This worked quite well and allowed us all to get work done on the project as well as see the progress of others in the group.