Ebola Crisis Data

Developing a Relational Database and User Interface
to Analyze Humanitarian Data and Tweets

Elements of Databases – Professor Cohen

Team: Justin Chao and Julianne Crea (JCrew)

Abstract

A relational database enables structured organization of data in tables that relate to one another through shared columns. This report details the construction of such a database in MySQL for data relating to the Ebola virus outbreak of 2014-2016, particularly in the countries of Guinea, Liberia, and Sierra Leone. Using Python, MySQL, and the Twitter API, the result of this report enables users to view and analyze collected data and Ebola-related tweets, allowing for a greater understanding of what caused an epidemic of such great magnitude.

Table of Contents

1. Introduction

1.1 Context

Ebola is an often fatal illness contracted through contact with the bodily fluids or organs of infected animals. The largest and most complex Ebola outbreak since the virus was first discovered in 1976 began in early 2014 and continued for two years. According to a Ebola factsheet published by WHO, this severe illness affected countries in West Africa— particularly Guinea, Liberia, and Sierra Leone. Ebola Treatment Centers (ETCs) were set up by a variety of organizations in order to care for people diagnosed with the virus. We have gathered Ebola-related data from various sources and organized it in a relational database as a set of tables. This relational design uses redundant data to link records in different tables (Beaulieu 14).

1.2 Motivation

As a team composed of a Chemistry major and a Mathematics major, we were interested in analyzing a health epidemic. According to data published by the CDC, the Ebola outbreak caused 2,544 deaths in Guinea, 3,956 deaths in Sierra Leone, and 4,810 deaths in Liberia between March 2014 and April 2016. There were more than 28,000 total cases in these countries during this time period. As announced in a press release published by WHO, the outbreak was not declassified as a Public Health Emergency of International Concern until March 29, 2016. With an epidemic of such magnitude, our main goal was to organize information about the crisis so that we could easily retrieve and analyze the data. Doing so facilitates investigation of factors that potentially assisted the spread of the virus.

2. Approach & Solutions

2.1 Data Collection

Our search for Ebola crisis data led us to "The Humanitarian Data Exchange" (HDX), a site where users associated with an organization can contribute. HDX is managed by the United Nations Office for the Coordination of Humanitarian Affairs (OCHA). We downloaded three csv files about Ebola Treatment Centers in Guinea, Sierra Leone, and Liberia. These were uploaded by the United Nations Mission for Ebola Emergency Response and gave geographic location, center name and status, number of beds open, and partner organizations, among other attributes.

Controlling an Ebola outbreak relies heavily on community engagement and on "raising awareness of risk factors for Ebola infection and protective measures" (WHO Factsheet). Thus, we downloaded data from a December 2014 survey conducted in Monrovia, Liberia's capital. The survey asked adult respondents in various communities about community outreach efforts, knowledge of Ebola symptoms and spread, prevention practices, and economic livelihoods. The dataset was uploaded by MIT Governance Lab, a research group of political scientists.

We also downloaded data about organizations involved in the response to the crisis, as these were mentioned in the treatment center csv. This set identified organizations, which country they were working in, associated acronyms, and organization type (government, private, etc.). The data was collected by OCHA's Regional Office for West and Central Africa.

Finally, a WHO Ebola factsheet states that Guinea, Liberia, and Sierra Leone all have "very weak health systems, lack human and infrastructural resources, and have only recently emerged from long periods of conflict and instability". We looked at each country on CIA World Factbook for more context. Using several economic and societal indicators, we created a csv file.

2.2 Data Modeling & Schema Design

2.2.1 Conceptual Model

Our first step before creating our database in MySQL was to model the data. This allowed us to identify our entity classes/tables, as well as the relationships between tables. We started with a conceptual model to show which entity classes relate. Fig. 1 shows our first attempt. We used LucidChart to draw a schema representing the basic structure of our database of four tables: “Survey_Respondent” for the Liberia survey data, “Country” for the World Factbook data, “ETC” for the Ebola Treatment Center data, and “Organization” for the partner organizations data. Within each entity class’s rectangle is a list of the attributes/columns in that respective table. For example, each record/row in the “Country” table contains a country name and that country’s health expenditures, GDP per capita, and urban population.

We can clearly see the tables’ relationships: a survey respondent must reside in a one country while a country may or may not have many survey respondents; a country may or may not have multiple organizations working within it while an organization must be within one country; a country may or may not have ETCs while an ETC must be in one country; an ETC may or may not have multiple partner organizations while an organization must assist at least one ETC. The ETC and Organization tables have a many-to-many relationship that must be resolved with a junction table, as we don’t want multi-dimensional data types i.e., lists, in a cell.

2.2.2 Logical Model

Our next step was to create a logical model providing constraints, data types, and primary/foreign key relationships. Fig. 2 shows this schema. We resolved the many-to-many relationship with a new table, “ETC_Org”, that has a composite key composed of an ETC’s code as given in the ETC table and of that ETC's partner organization's name as given in the Organization table. These two attributes are primary keys in their respective tables. The new ETC_Org uniquely identifies ETCs that had partner organizations or organizations that were assisting an ETC. With the junction table, our model is in first normalized form with scalar values in all cells and we can query information about an ETC and its partner organization.

We also identified which attributes in each table are a primary or foreign key. A primary key "uniquely identifies a row in that table" and cannot be null (Beaulieu 14-15). A foreign key signifies which column to refer to in the linked parent table. In the Survey_Respondent table, a respondent's ID uniquely identifies them. Their country of residence is a foreign key and references economic indicators about that country in the Country table. In the Country table, a country's name uniquely identifies it. In the Organization table, an organization's name is its unique identifier and its home country references country_name in the Country table. Finally, a treatment center's code number uniquely identifies it in the ETC table and its country references country_name in the Country table. We can easily query information about a respondent and their country, and about a treatment center or organization and the country they are in.

The logical model shows each attribute's data type. We used char, varchar, double, and integer types. "Double" handles decimals and is used for geographic coordinates and two economic indicators given as percentages. "Char" holds a fixed length string with the size in parenthesis. "Varchar" holds a variable length string with the maximum size in parenthesis. We used "char" for etc_code (all are eight digits long) and for lab_present and gender, which take one-letter values: Y/N or M/F.

2.2.3 Amendments

As we progressed with our database design, we realized we had made some errors in our schemas. With only the ETC and Organization tables, we were limiting our data to include only the organizations that had helped one of the given ETCs—and excluding other organizations that responded to the crisis but helped at other locations. We created another table called "Partner_Orgs" which contained organizations that *did* help one of our listed ETCs. We then created a junction table between this and the ETC table to resolve the many-to-many relationship. We also realized that an organization could operate in multiple countries, so the Country-Organization relationship became many-to-many. We resolved this with a junction table with a composite key consisting of country_name and org_name to refer to the two parent tables. The new schemas can be seen in Fig. 3 and Fig. 4.

2.3 Data Loading

Prior to loading our data, we created a data dictionary and combined the three files about Ebola Treatment Centers in Guinea, Sierra Leone, and Liberia into one csv file. We also researched various organizations that were missing data in the "acronym" and "type" columns to discover those attributes and reduce the sparseness of the "Organization" table.

Work on our actual database began with a "Create Table" script in MySQL. We started by dropping the database "Ebola" if it existed and then creating it and using it. Next, we dropped the tables Country, ETC, Partner_Orgs, Partner_Org_ETC, Survey_Respondent, Organization, ETC_Org, and Country_Org if they existed. Then we wrote create table statements for each of these tables, identified column names and their data types, set primary and foreign constraints, set not null constraints, and identified which attributes and parent tables the various foreign keys referenced. Some of these statements also ensure that a cell has a certain value (example: the gender column must be “M” or “F”). The “Create Table Country” statement can be seen below.

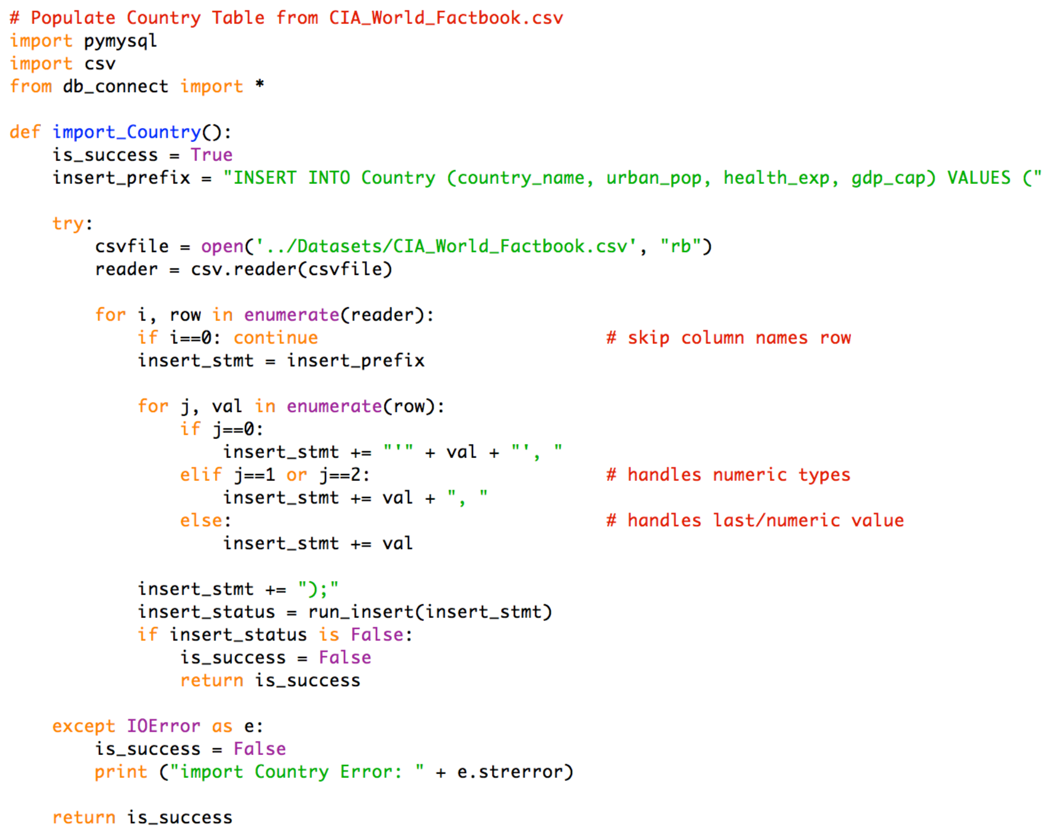
```
CREATE TABLE Country (
    country_name        VARCHAR(100)    NOT NULL,
    health_exp          FLOAT           NOT NULL,
    gdp_cap             INTEGER         NOT NULL,
    urban_pop           FLOAT           NOT NULL,
    PRIMARY KEY (country_name)
);
```

Figure: A portion of the create_tables.sql script

Next, we created the db_connect.py file, which started by importing PyMySQL. PyMySQL is used to connect to a MySQL server from Python. We wrote a function create_connection() that connected to the server and returned any existing error messages. A destroy_connection() function closed that connection. The main function here is the run_insert(insert_stmt) function, which takes a SQL statement as a parameter, connects to the server, executes the statement in MySQL and then closes the connection.

Our next step was to write rollback scripts removing all data from each table. These started by importing PyMySQL and the functions from our db_connect.py file. The scripts use the run_insert(insert_stmt) function to run the SQL command “DELETE FROM *tableName*”. The rollback script for the Country table can be seen below.

Finally, we loaded the data from our csv files using an import script for each table. This used PyMySQL, db_connect functions, and the csv package— which lets Python to read and write data in CSV format. These scripts have a hard-coded statement of the form “INSERT INTO *tableName (attribute1, attribute2, …)* VALUES (”. The script opens a csv file and reads it in as an array. It iterates through each row in the csv, skipping the first one with column names. As it goes through the remaining rows, it considers each cell. If the cell is a string, it adds “,” around the value and concatenates this to the insert statement. If the cell is a numeric type, it adds a comma after the value and concatenates this to the statement. For the final column value, it doesn’t add a comma but concatenates the value to the statement. A last line adds); to the statement so that MySQL will recognize the statement’s end. The run_insert(insert_stmt) function executes this statement in the database. The code for the Country table is below.



```
# Populate Country Table from CIA_World_Factbook.csv
import pymysql
import csv
from db_connect import *

def import_Country():
    is_success = True
    insert_prefix = "INSERT INTO Country (country_name, urban_pop, health_exp, gdp_cap) VALUES ("

    try:
        csvfile = open('../Datasets/CIA_World_Factbook.csv', "rb")
        reader = csv.reader(csvfile)

        for i, row in enumerate(reader):
            if i==0: continue                           # skip column names row
            insert_stmt = insert_prefix

            for j, val in enumerate(row):
                if j==0:
                    insert_stmt += "'" + val + "', "
                elif j==1 or j==2:                      # handles numeric types
                    insert_stmt += val + ", "
                else:                                   # handles last/numeric value
                    insert_stmt += val

            insert_stmt += ");"
            insert_status = run_insert(insert_stmt)
            if insert_status is False:
                is_success = False
                return is_success

    except IOError as e:
        is_success = False
        print ("import Country Error: " + e.strerror)

    return is_success
```

Figure: Import Script for Country

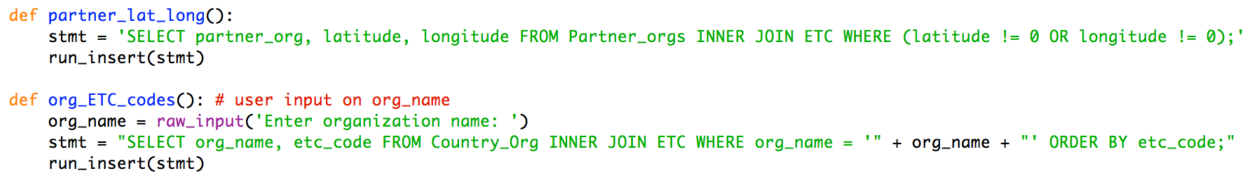
A master script called populate_database.py imports and runs the rollback script for each table and then runs the import script for each table. As a result, the database is populated and the user is told whether these operations succeeded or failed.

2.4 Query Interface

2.4.1 Queries

In order for a user to easily view and analyze data about the Ebola crisis, we developed a query interface using Python. Our first step was to write a query_functions.py program in Python containing select SQL statements that a user can choose to run. This script again uses PyMySQL and the db_connect.py functions. It contains seventeen functions: ten that execute a query on our base tables, two that create views, and five that execute a query of those views. Some of these queries allow a user to include their own input as to what will be queried, others do not.

For those queries that do not allow user input, the defined function is simple. A SQL select statement is hard-coded and assigned to the variable stmt. The function then calls the now familiar run_insert() function from db_connect, executing the statement in MySQL. For queries that do allow user input, the raw_input function is used to prompt the user to enter their desired input and their response is assigned to a variable. That variable is then inserted into a SQL select statement and executed in MySQL. Examples of both of these query types are shown below. The first pulls organizations from the Partner_orgs table and inner joins them with the coordinates of the ETC that the organization helps. It only pulls ETCs whose coordinates are not 0. The second requires a user to input an organization name and then inner joins that with ETCs that that particular organization has helped, ordering the results by etc_code.



```
def partner_lat_long():
    stmt = 'SELECT partner_org, latitude, longitude FROM Partner_orgs INNER JOIN ETC WHERE (latitude != 0 OR longitude != 0);'
    run_insert(stmt)

def org_ETC_codes(): # user input on org_name
    org_name = raw_input('Enter organization name: ')
    stmt = "SELECT org_name, etc_code FROM Country_Org INNER JOIN ETC WHERE org_name = '" + org_name + "' ORDER BY etc_code;"
    run_insert(stmt)
```

Figure: Sample Queries

The queries created allow a user to analyze the Ebola crisis data. A user can choose to view ETCs from a chosen country with a user-defined minimum number of open beds, showing how the centers have been filled with patients. Other queries display the average age and average education level for males or females in a given country, allowing for interpretation as to how education levels may affect Ebola transmission. One of the views we created helps to protect survey respondents as it hides their identification from the user, only displaying their gender, age, education, and country of residence. In addition to this Python script containing the queries, we also wrote a SQL file containing the same select statements and create view statements.

Sometimes queries can be maliciously manipulated and inject harmful SQL, such as a statement that deletes a table. Thus, we included a run_protect(statement) function in our db_connect.py script. This helps to protect against SQL injections. Essentially, if a statement contains phrases like “drop table”, “truncate table”, “delete from”, or “or 1=1”, the statement will not be executed. This is to protect the tables that have already been created from being deleted or having their data cleared.

2.4.2 Interface Creation

The actual creation of the interface occurs in a Python program that imports PyMySQL, db_connect functions, and functions from the query_functions file discussed above. We first defined the function print_menu(), which prints the various query options a user can choose from in a numbered list format. A run_query_case() function lets a user input the query option number they want to run. The script then uses if, elif, and else statements to determine which function from query_functions.py to run, thus executing the SQL query statement. After running the function, another function called run_another() is executed. This asks the user if they want to run another query, and accepts and responds to an answer of either “Y” or “N”. If the response is

"Y", run_query_case() is called again. If the answer is "N", the interface prints a goodbye message. Additionally, the run_query_case() function has a safeguard against a number being entered that isn't included on the menu options. If such an event occurs, the interface informs the user that the number is not valid and prompts them to enter another number. This interface provides users who do not have the requisite technical skills to view and analyze data.

2.5 Twitter API

2.5.1 Extend Database

As a final feature, we included the ability to pull tweets from the Twitter API and store them in a table so that users can analyze Ebola-related tweets. First we extended our database by creating a Tweet table. The table takes items from the JSON tweet_doc pulled from the Twitter API and stores them in columns as a varchar, integer, or datetime data type. We use json_extract(tweet_doc, '$...'), where the $ refers to the tweet_doc and the statement that comes after the $ identifies which part of the JSON document to extract. For example,

"screen_name varchar(32) generated always as (json_unquote(json_extract(tweet_doc, '$.user.screen_name'))) stored"

removes the Tweet's author's screen name and stores it in a column called screen_name as a string. The create table statement also makes columns for the tweet ID, follower count, the tweet's creation timestamp, retweet count, language, origin, the tweet's text, and the country mentioned in the tweet. This country name is defined as the foreign key and references the country_name attribute in the Country table.

2.5.2 API Client

To pull tweets, we created a Twitter API app online and got the API keys and tokens for access. These are stored as variables in an api_client.py file, which collects the tweets. In this program, we have a function get_api_instance() that uses the tweepy package to get access to the Twitter API. Another function called do_data_pull(api_inst) uses the connection to the API initiated by the previous function. We hard-coded the SQL query "SELECT country_name from Country order by country_name" to return the names of the three most affected countries during the crisis. For each of these, a search is sent to the Twitter API requesting tweets that include country name in the tweet content as well as "#Ebola". The program then inserts the JSON-type tweet_doc and the country name into the Tweet table in our database.

2.5.3 Tweet Queries

We wanted to allow users to easily analyze tweets as well, so we added queries accessing the new Tweet table to our query interface. We began by adding nine new functions to the query_functions.py file, formatting them the same way we did for queries accessing our base tables and views. These queries allow users to see how many tweets there were about Liberia, to order tweet IDs and screen names by retweet count or by follower count, to view tweets from countries who spent a certain amount on health expenditures, and to view all of the pulled tweets about either Guinea, Liberia, or Sierra Leone. These queries were also added to a json_queries.sql file. The new options were added to the interface menu and their functions called in the extended_query_interface.py file. They appear and can be used in the interface.

3. Conclusion

3.1 Main Result

We were able to create a database on the MySQL server that creates tables to contain Ebola-related data downloaded from a humanitarian data exchange. This relational database links tables by primary and foreign keys assigned to shared attributes, creating an organized structure. The database also contains tweets pulled from the Twitter API that contain either the term “liberia”, “guinea”, or “sierraleone” in conjunction with “#Ebola”. This enables users to view and analyze what is being said about the Ebola virus in a given country on social media. Users who understand SQL can work directly in the MySQL environment, creating queries to analyze data at will. For those who do not know SQL, we have created a user-friendly interface with pre-composed queries displayed in a menu. These allow users to work with the data, though it limits what they can analyze.

As we discovered while adding our Tweet queries, it is easy to add more queries to the interface menu. Thus, users could request an option to be added and it could be done without much extra effort, expanding their analysis capabilities. Additionally, the API client can easily be customized to search for tweets that contain a country’s name and another hashtag, or anything that the user desires to search for.

3.2 Technical Challenges & Lessons Learned

Although the results are easy to customize, we did encounter some minor technical challenges when creating the database, interface, and API client. Some of these were as simple as learning to work in the Terminal environment and using Github for the first time to work on the same project remotely without encountering commit/merge issues. We learned how to work in both of these environments, and how to write a variety of SQL queries— thus gaining a better understanding of the language. Additionally, this was our first exposure to working with an API and gaining access to public tweets. This is an interesting skill to work with in the future if we wish to investigate the language being used to discuss other topics on social media.

Two of our more involved challenges were our logical model and the formatting of query outputs. As discussed in section 2.2 (Data Modeling & Schema Design), we had to add a second junction table to resolve the relationship between ETCs and partner organizations. We were initially thrown off by how this made our logical model look circular and redundant, but it was confirmed that this was the correct way to normalize our structure. Figuring out how to format the query outputs in our interface was also a challenge, and has not been entirely resolved. For some of the queries that return large data, the formatting does not look entirely correct. This will require us to look into Python formatting options in more depth. Regardless, our interface functions and accomplishes what it was intended to—allowing users to analyze Ebola crisis data.

4. Bibliography

"2014 Ebola Outbreak in West Africa – Case Counts." *Centers for Disease Control and Prevention.* Centers for Disease Control and Prevention, 14 Apr. 2016. Web. 06 Dec. 2016.

Beaulieu, Alan. *Learning SQL.* N.p.: O'Reilly Media, 2009. *ProQuest Ebook Central.* Web. 5 Dec. 2016.

"Ebola Virus Disease." *World Health Organization.* World Health Organization, 18 Jan. 2016. Web. 06 Dec. 2016.

World Health Organization. Emergency Committee. *Ebola Is No Longer a Public Health Emergency. World Health Organization.* World Health Organization, 29 Mar. 2016. Web. 05 Dec. 2016.

5. Additional Figures

Figure 1: Conceptual Model (First Attempt)

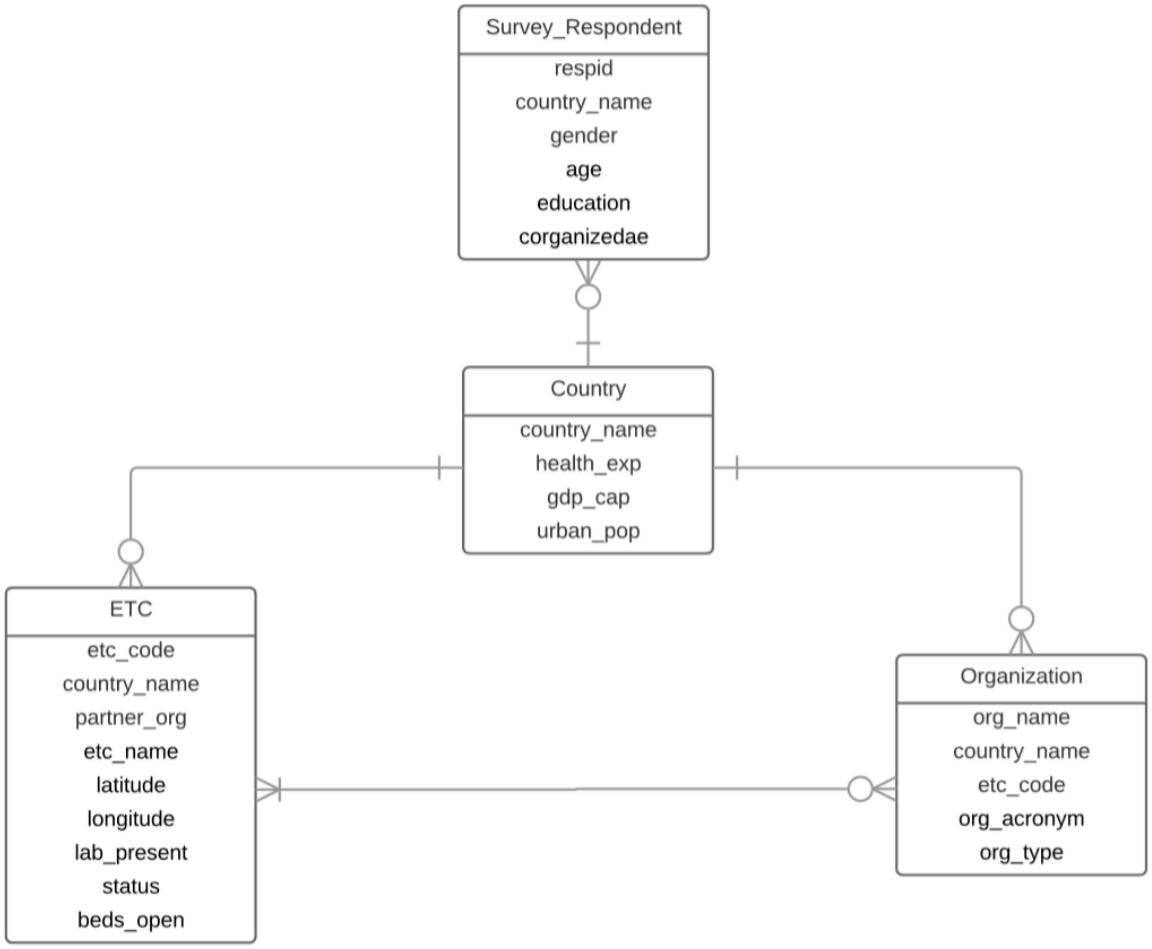


Figure 2: Logical Model (First Attempt)

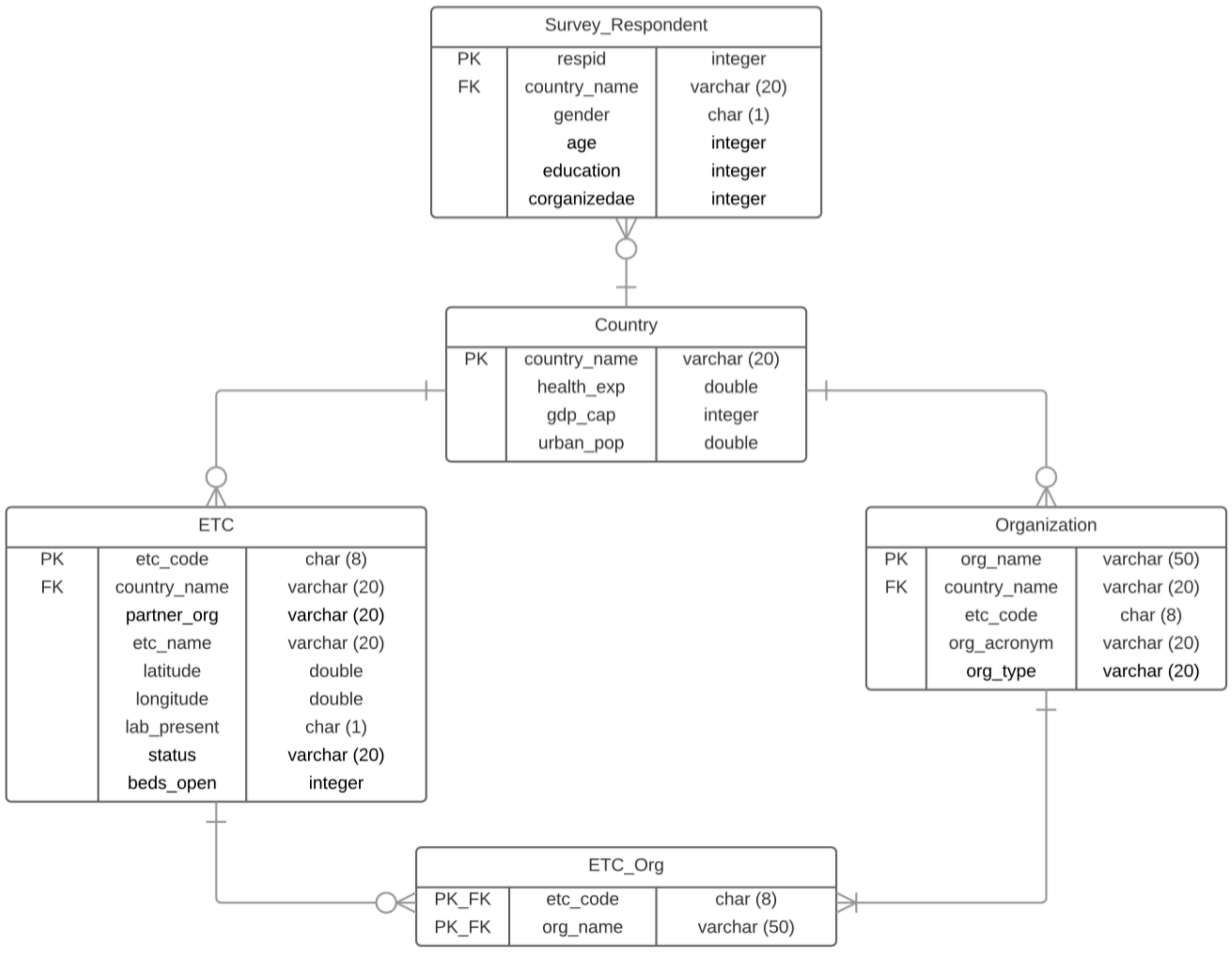


Figure 3: Conceptual Model (Final)

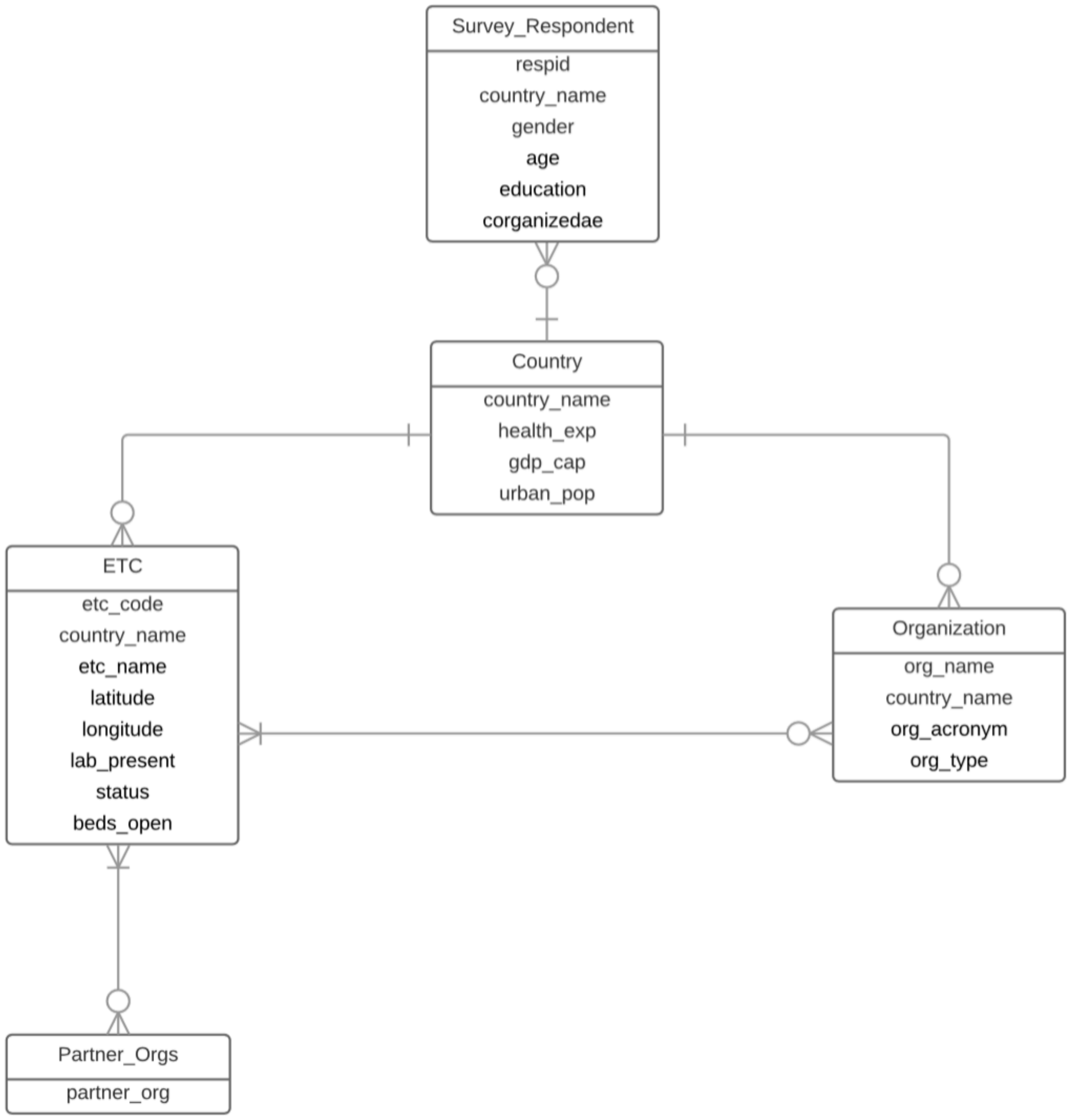
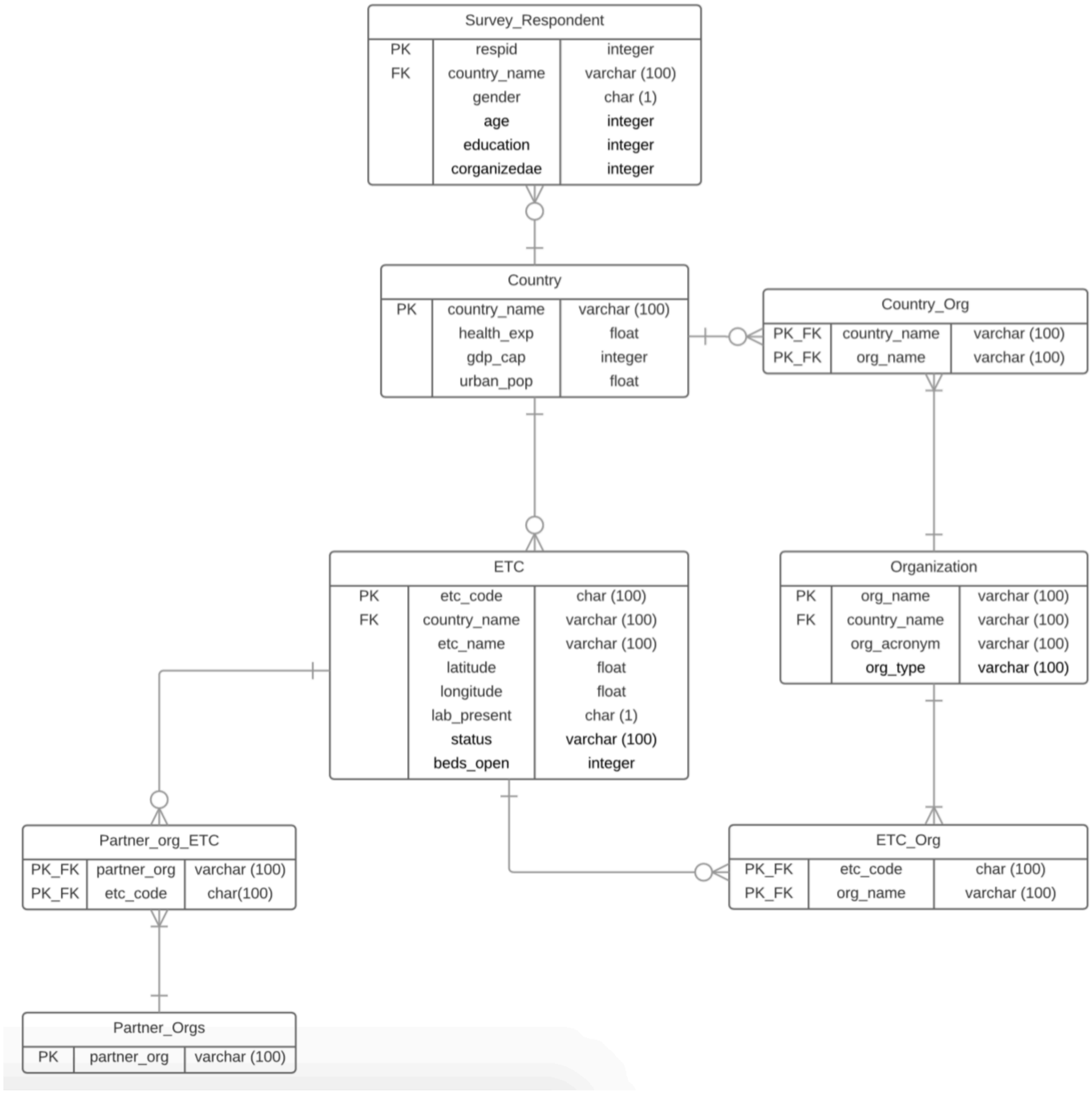


Figure 4: Logical Model (Final)



Survey_Respondent

PK	respid	integer
FK	country_name	varchar (100)
	gender	char (1)
	age	integer
	education	integer
	corganizedae	integer

Country

PK	country_name	varchar (100)
	health_exp	float
	gdp_cap	integer
	urban_pop	float

Country_Org

PK_FK	country_name	varchar (100)
PK_FK	org_name	varchar (100)

ETC

PK	etc_code	char (100)
FK	country_name	varchar (100)
	etc_name	varchar (100)
	latitude	float
	longitude	float
	lab_present	char (1)
	status	varchar (100)
	beds_open	integer

Organization

PK	org_name	varchar (100)
FK	country_name	varchar (100)
	org_acronym	varchar (100)
	org_type	varchar (100)

Partner_org_ETC

PK_FK	partner_org	varchar (100)
PK_FK	etc_code	char(100)

ETC_Org

PK_FK	etc_code	char (100)
PK_FK	org_name	varchar (100)

Partner_Orgs

PK	partner_org	varchar (100)