**Event Data TwoRavens**

**Documentation and User Manual**

**Table of Contents**

* **Introduction**
* **Setting up**
* **Implementation**
  + **REST Calls**
  + **Current Design and backend**
* **Proposed Design**
* **Resources**

**Introduction:**

**TwoRavens** is a system of interlocking statistical tools for data exploration, analysis, and meta-analysis. The first to be released is an interface for quantitative analysis, that allows users at all levels of statistical expertise to explore their data, describe their substantive understanding of the data, and appropriately construct statistical models. This integrates with Dataverse [Project](http://dataverse.org)|[GitHub](https://github.com/IQSS/dataverse)) and Zelig ([Project](http://zeligproject.org)|[GitHub](https://github.com/IQSS/Zelig)), through a portable, lightweight, browser-based and gesture-driven interface, allowing users to run statistical models available in Zelig on data archived in Dataverse.

The project page is available at <http://2ra.vn>

Usually, TwoRavens has been created to work with tabular data of smaller scales. But, the problem we try to tackle with Event Data TwoRavens is to give the user tools to browse through large datasets, primarily consisting of event coded data. Event Data is data coded from news events, curated through either news websites, or archived print media news reports. The four main components of a typical event data dataset are **Actor**, **Location**, **Date** and **Event type**. Let us explain the four briefly:

**Actors:**

**Location:** The location signifies the place where the event in question occurred. The granularity of the location can be anywhere from the country where the event took place, to the locality of the occurrence of the event.

**Date:** The date on which the event occurred.

**Event Type:**

**Where does TwoRavens come in:** A typical Event Data dataset contains millions of records of coded events, which grow by the day as and when new news reports are being coded.

TwoRavens provides a way to easily select and analyse that data using its familiar GUI architecture, and giving the user control over what the user wants to see, what he/she wants to do with it.

There are potentially three functions TwoRavens aims at implementing in analyzing EventData:

**Subset**: As there are millions of entries in the dataset, we have to provide the user with tools to select and filter data according to their requirement. The User can select data according to a specific source actor, target actor, location, date and event type, to start with. The user can do what he/she wants with the selected data. He/She can download the data, or use the TwoRavens Architecture to further analyse the data.

**Aggregation**: Even if the user has selected data at the subset stage, the data is still raw, and cannot be analyzed at that stage. TwoRavens provides tools to count the events according to a temporal unit(date) or spatial units (actors, locations). The choice of the aggregation unit is completely on the user.

**Analysis**: After the data is aggregated, it is ready to be analyzed using time series analysis. The analysis models are implemented into TwoRavens, giving the user the option to customize his/her models and then estimate results.

**The Project**

**Setting up**

To start working on TwoRavens, you need to set it up first. The installation steps are very easy, and you will need very little time to do that.

Requirements:

* Github account, and preferably a github GUI client(For windows) or git installed(for linux or macOS)
* R software, and RStudio. R can be downloaded on <https://www.r-project.org>. RStudio can be downloaded from <https://www.rstudio.com/products/rstudio/download> Note: Rstudio is an IDE for R, so it’s your choice to download it.
* Python 2.7. It can be downloaded from <https://www.python.org/downloads/release/python-2712/>

Note: Linux usually has python installed. Use python -V to see if python is installed. For Windows users, make sure you check the option “add python.exe to system path” during installation.

Additional notes: Windows blocks cross domain requests between client scripts and server even if CORS(cross-origin-resource-sharing) is enabled in the request and response headers. And because the R server serves on a different port than the python server, it is very important to enable cross origin resource sharing. Linux based operating systems and macs do not have this problem. So, it is suggested to set up the project on a virtual ubuntu machine if you have windows as your primary OS. You can download Oracle virtual box from <https://www.virtualbox.org/wiki/Downloads> and ubuntu from [www.ubuntu.com/download](http://www.ubuntu.com/download) .

Once you have all of the above, do the following steps to setup tworavens locally:

1. Open the terminal/command prompt, and type

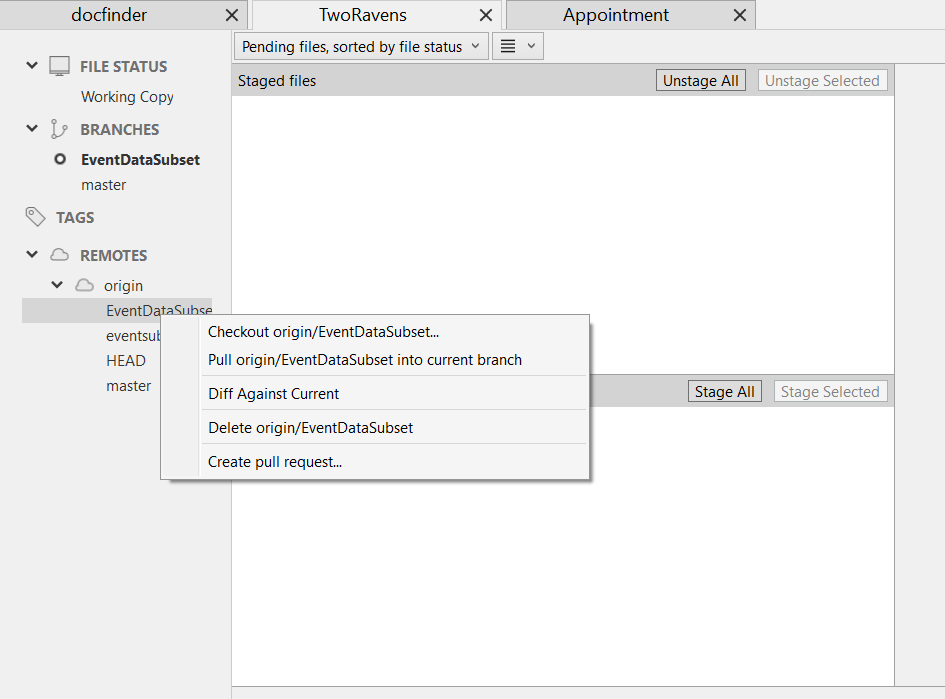
git clone <https://github.com/vjdorazio/TwoRavens.git>

if you are using a github GUI like source tree or smart git,, click on clone/new repository, and enter the above URL when prompted.

1. Once you have cloned TwoRavens, you will need to change the branch from master to EventDataSubset. This branch contains the codebase for Event Data TwoRavens. To do that, in the terminal/command prompt, type

git checkout EventDataSubset.

For GUI clients, look for branches under origin, right-click on the EventDataSubset branch, and select checkout.



1. Open R,either by typing R into the terminal for mac/linux users, or by selecting the R program for windows, and Execute the following with R to install R packages:

install.packages(c("VGAM", "AER", "dplyr", "quantreg", "geepack", "maxLik", "Amelia", "Rook","jsonlite","rjson", "devtools", "DescTools", "Zelig"))

1. After the installations are done, set the working directory to TwoRavens/rook. For example,

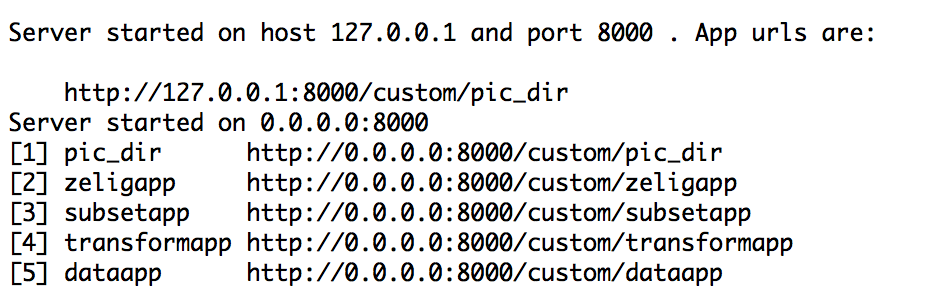
setwd(“/Users/vjdorazio/Desktop/github/TwoRavens/rook”) for linux/mac or

setwd("C:/users/rohit/documents/TwoRavens/rook") for windows.

1. To run the R server, type

source(“rooksource.R”)

This will give a large output, as it will install some more dependent packages when you run this for the first time. The final output should look like this:



Note: Try doing this in the console provided by R, and not Rstudio. R studio uses the default 8080 port for its own purposes sometimes, so it might throw an error that the port is already in use. You can use RStudio for editing the R scripts, but try running this on the R console.

1. Once this server is up and running, launch the terminal/command prompt, and navigate to the TwoRavens directory. For example,

For linux/mac users: cd Desktop/github/TwoRavens

For windows users: cd C:/users/rohit/documents/TwoRavens/rook

After navigating there, you will need to start a server, which we will do using python. Type in the following command:

python -m SimpleHTTPServer 8888 &

This should give an output: Serving HTTP on 0.0.0.0 port 8888 ...

1. Once this is up and running, you are setup! Go to the browser, and type in <http://localhost:8888/gui3.html> to open the TwoRavens Event Data page.

**Implementation:**

We only focus on the Event Data part of two ravens. We start with the backend of the application, which is implemented in R. The R scripts are again divided into subset, aggregation and analysis.

**Subset:**

**RESTful Call and MongoDB database:**

The main dataset is hosted as a mongoDB on the utdallas server, and is accessed through a RESTful api. The database is queried using mongoDB queries in JSON format, which is appened into the REST url call.

The database has been hosted as a web server for two reasons:

1. This is a huge and ever-growing database, with thousands of events coded into the dataset every day. It is not possible to keep everything on the local systems due to limitations of both size, and access time.
2. Hosting it on a web server helps us modularize the TwoRavens to work with not only the Phoenix dataset we are working with now, but with other event datasets like IQSS dataset in the future. As we are accessing the web service using a simple url call, we just have to change the url and queries to get relevant results from other hosted web services.

Currently the phoenix mongoDB server is being handles by Sayeed Salam (email: [sxs149331@utdallas.edu](mailto:sxs149331@utdallas.edu)), who is an RA for Dr.Latifur Khan, and is working on the NSF grant which is encoding the phoenix dataset, among other things. He will be your point of contact for all Phoenix dataset and mongoDB access Queries. The data-server is called the spec-event-data-server, and the github link for the project is given below:

<https://github.com/Sayeedsalam/spec-event-data-server/blob/master/README.md>

This link gives a readme about how to set up the URL calls, and the basic query structure in detail. I will still add a brief description of how to set up the url call below:

1. The REST url is of the format

[http://10.176.148.60:5002/api/data?api\_key=<api-key-here>&query=<query-for-data>&select=<comma-separated-list-of-attributes>&unique=<name-of-attribute](http://10.176.148.60:5002/api/data?api_key=%3capi-key-here%3e&query=%3cquery-for-data%3e&select=%3ccomma-separated-list-of-attributes%3e&unique=%3cname-of-attribute)>

1. The API key will be given by Sayeed**.** Contact him at the given email address for the api key.
2. Let’s talk about the list of attributes first. Here are all the attributes in the current hosted web server:

'event\_id','date8','year','month','day','source','src\_actor','src\_agent','src\_other\_agent','target','tgt\_actor','tgt\_agent','tgt\_other\_agent','code','root\_code','quad\_class', 'goldstein','geoname','country\_code','admin\_info','id','url','source\_text','longitude','latitude'

I will briefly elaborate the ones we use, or might need in the future:

1. Event\_id: The unique id given to every event
2. Date8: the date of the event, in the format yyyymmdd, example: 20150526
3. Year, month, day: the event date broken up into year, month and date, so that we can subset using just year or just month easily.
4. source: the full actor code of the source actor. The code ranges from 3 to 12 characters, and are in sets of three. For example, a source actor can be USAGOVGOV, which is formed using three parts: the source actor(USA), the source agent(GOV) and the source-other-agent(s) (GOV). This is a hierarchical model, and almost all events have at least the source actor, and the source agent. Also, the source-other-agent can 3,6 or even 9 characters. But, the actor and and agent are always 3 characters each. The source is also divided into actor, agent and other agent columns after source to simply get just actors or agents if needed.
5. Target: It has the same code structure as source, but for the target actors.
6. Root\_code: root code gives the type of the event. It ranges from 01 to 20, which are 20 different categories in which an event can be coded.
7. Country\_code: the ISO code of the country in which the event happened. Right now, the country\_code also has some coded which are not for any country. Right now, the country\_code just has location names, and not only country names. But, that is not our issue, that will be fixed by sayeed’s team.

The rest of the attributes/columns are not really important right now, as they are mostly blank.

1. For queries, the main structure is

{attribute:value}.

For example, if I want to get events for 26th may 2011, my query will be,

{“date”:”20110526”}.

For suppose source actor is usa, the query would be

{“src\_actor”:”USA”}.

Note that both the attributes and the data is case-sensitive.

Also, in locations, if the column is there, there might by spaces, which will result in as error, as urls cannot handle space.

You have to replace all spaces with **%20%** before executing the url. Hence United states of America will become United%20%states%20%of%20%America.

If you want to do an or search, the query will be:

{“$or”:[{“Attribute1”:”Value1”},{“Attribute2”:”Value2”}]}

For example,

{“$or”:[{“source”:”USAGOV”},{“source”:”RUSMIL”}]}

Notice the use of square brackets to deifne a list in JSON format.

If you want to give an and condition, the query is:

{“Attribute1”:”value1”, “Attribute2”:”value2”}

For example

{“source”:”USAGOV”,“source”:”RUSMIL”}

If you want to specify that the value should be in a list, for example, the country should be from india, usa, china, russia and japan, us use $in:

{“country\_code”:{“$in”:[“IND”,”CHN”,”USA”,”JPN”,RUS]}}

Greater than is given as $gt, and less than is given as $lt. So, if u want a date to be after 1st january 2010 and before 31st december 2011, the query will be:

{“date8”:{“$gt”:”20100101”,”$lt”:”20111231”}}

These are the main queries which are required right now, for more queries, and operators, refer to the link given above, or go to google and search for mongoDB commands.

1. After queries, there is the select operator. This operator is like the SELECT operator in traditional database applications, as in it filters out which attributes you would want returned. So, if you want source, date, target and location, you would give &select=source,date8,target,country\_code.
2. Sayeed has also baked in the unique functionality, which is very important for filtering out repeated data. We give our users some information first to let him/her select the data he/she wants. For that, we need a list of data for all the attributes we have. This list should just have all data once, as we are just listing out all the unique data present for that attribute. Unique lets us do that. If we want all unique source, we write &unique=source.

This would return a list of all unique sources present in the dataset. Note that unique nullifies the select operator, as unique will always send one list of one attributes, and nothing else. Also, u cannot use unique as &unique=source,target, as it would return blank, or an error, as it only sends unique data about one attribute. If you want to get unique data for multiple attributes, you must write a new query for each attribute.

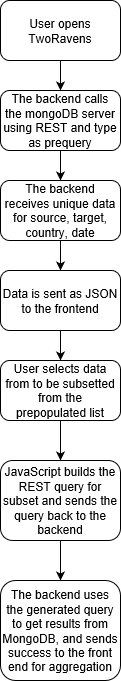
1. Few additional notes about the REST call:
   1. The web service is hosted at the UTDallas servers, and you need to be connected to a UTD network to access the server. If you work from the niversity, just connect to cometnet and you will be fine. If you are working from home, you will need to connect to the UTD VPN. If you have not done that before, go to this link ( <http://www.utdallas.edu/oit/vpn> ) which has all the necessary steps and softwares you need to connect to the VPN
   2. The server might shut down or crash from time to time, but in most cases, it can be restarted. You can contact Sayeed on how to restart the server by yourself.

You can resume the REST calls after the server restarts.

* 1. If there are still errors in the server, take a screenshot of the error and send it to Sayeed, and he will resolve it. He usually does it within a few hours, but might take longer. If you don’t see anything by the next day, follow up with him, and keep prof. Vito in the loop.
  2. Note that only unique function of mongoDB has been added to this database, and not min or max or any other functionalities. You can ask Sayeed to add these functionalities, but if not, R is more than capable of handling these functions.

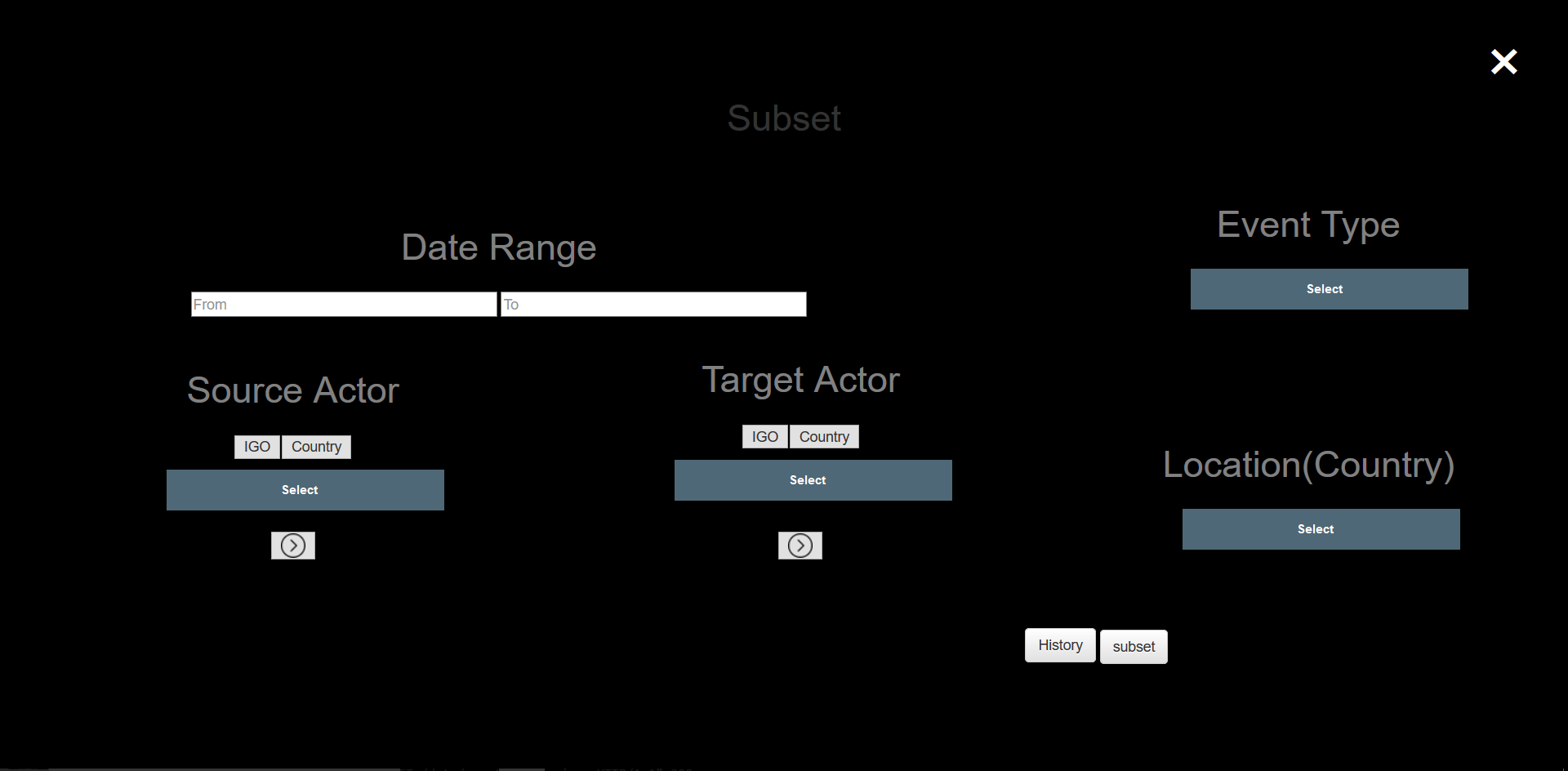
**Current design and Backend**:

Now that you know about the REST calls, and query, let us discuss the workflow of the subset process. Below is the flow chart to give you a brief overlook:



* If you look at the flow chart, most of the heavy lifting is being done at the web service, and we just have to build and send the query through REST.
* The interface of subset has gone through some different designs. At first, we implemented an overlay which sits on top the standard TwoRavens interface, which had dropdown menus, and sliders for the user to select data to subset.

This was the previous interface:



* This interface was made as a prototype or a proof of concept to showcase the ability of our backend to query the Phoenix server and get data according to the user’s selection. As you can see, this interface gave the user the option to select source, target, event type(root\_code), location and the date range of events.
* As you will see, this is very different from the traditional TwoRavens design. This was more of a placeholder interface, to show something for the NSF grant conference in January 2016.
* Even if we are not using this interface, this interface still exists as gui2.html and app\_ddi2.js. This interface has not been overwritten as the JavaScript code has the whole query building function, which you will need to migrate to the new interface.
* For subset, the R script is not very important as it acts as a middle ground between the mongoDB server and the frontend. The R script (rookquery.R) only has code to define URLs and sending and receiving the data. The variable definitions and functions used are documented in the script.
* The main query building happens in the javascript. app\_ddi2.js has a function named callquery, which builds the query for both populating the selection lists, and the final query after the user selects the data.
* For populating the selection lists, the query is simple, we just put in type as “prequery” and send it to the backend. This happens in the prequery() function in JavaScript. The JSON which goes to R is below:

{“type”:”prequery”}

* The R script has a switch case construct on the type variable, which checks to see whether the type is prequery or postquery. If the type is prequery, it queries the database to get unique lists of source actors, target actors, location and date. As event type is a static list of 1 to 20, we can create that on the javascript side, and not query it.
* It takes around 10-20 seconds to populate all the lists, depending upon the size of the lists, and the bandwidth of the server. This is because of the growing size of the mongoDB database, which may require sending a lot of data from their server to the TwoRavens server.
* After the lists are populated, the maximum and minimum date in the returned list is saved, the locations, source and target actors are arranged alphabetically, and sent to the frontend in JSON format. The format is something like this:

{“maxdate”:”20150831”,

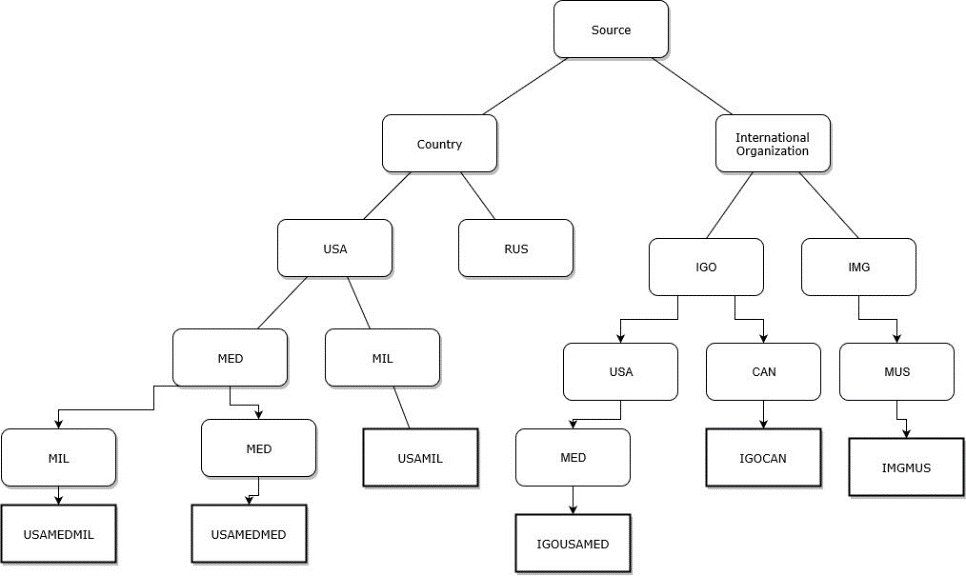
”mindate”:”20100101”,

”location”:[“usa”,”india”…],

“sourceactors”:[“USA”,”RUS”,”CHN”,”IND”…],

”targetactors”:[“SYR”,”AFG”…]}

* There are a few things to note here:
  + The source and target actors follow a hierarchy: - for every source/target actor, there is a source/target agent. For every agent, there can be sub-agents. This diagram below gives an idea:



* + For source and target actors, as it is a hierarchical list, we do not query the complete source code at once. Querying the complete actor code seems counter-intuitive as we have to separate the first three characters, and the next three characters ourselves, which is an added effort, and not necessary, as the separated characters are already there in the database as source\_actor and source\_agent.
  + The option of choosing these three levels is given in app\_ddi2.js, as three level drop downs. In the first level, the user chooses and international organization or country. After this selection, the list is populated with country names, or international organisations. The user then selects one of multiple actors from the populated list, and clicks on next. The code then sends the selected actors to R, which then queries the actors, and gets agents for each actor. That list is shown to the user, who selects agents for each actor.
  + For the actors, we first populate the list with the source/target actors (the first three characters in the complete code), then when the user selects a list of actors, we query the database again for the source/target agents (the next three characters of the complete actor code). We don’t go beyond the agent, as the rest of the code is not always present, and they have variable character lengths.
* Once the lists are populated and shown to the user, the user is free to select any of the five given options to subset the data. Once he is done selecting, the control goes to the function callquery(), which builds the query. The query building is simple, as JavaScript makes it very easy. To create JSON data, create an associative array, and keep adding the lists. For example:

var query={};

query[“type”]=”postquery”;

query[“query”]={“date8”:{},”country\_code”:{},”source”:{},”target”:{}};

query [“query”][“date8”][“$gte”]=selectedfromdate;

query [“query”][“date8”][“$lte”]=selectedtodate;

query [“query”][“country\_code”][“$in”]=countrylist;

query [“query”][“source”][“$in”]=selectedsources;

query [“query”][“target”][“$in”]= selectedtargets;

If the user leaves a list empty, we would need to remove that value from the query, as this is an and query for all attributes. We delete a key from the array just by using

delete query[“query”][“country\_code”];

After the associative array is complete, use

jsonout=JSON.stringify(query);

to convert the associative array into json.

* As mentioned before, we use cross-origin-resource-sharing (CORS) request to send data to the backend, as AJAX doesn’t support cross domain data transfer. the function callquery() calls the CORS request function, and handles and the success and failure of the request. A sample is given below.

function callquery(){

//code goes here

urlcall=< url of the script you want to send data to>

solajsonout=<data to be sent in JSON format>

function downloadSuccess(btn, json)

{//code if data transfer is successful }

function downloadFail(btn)

{//code if data transfer has failed }

makeCorsRequest(urlcall,btn, downloadSuccess, downloadFail, solajsonout);

}//end of function

The callquery function is calling the CORS function, which is given below:

function makeCorsRequest(url,btn,callback, warningcallback, jsonstring) {

var xhr = createCORSRequest('POST', url);

if (!xhr) {

alert('CORS not supported');

return;

}

// Response handlers for asynchronous load

// onload or onreadystatechange

xhr.onload = function() {

var text = xhr.responseText;

//console.log("text ", text);

try {

var json = JSON.parse(text); // should wrap in try / catch

var names = Object.keys(json);

}

catch(err) {

estimateLadda.stop();

selectLadda.stop();

//console.log(err);

alert('Error: Could not parse incoming JSON.');

}

if (names[0] == "warning"){

warningcallback(btn);

alert("Warning: " + json.warning);

}else{

callback(btn, json);

}

};

xhr.onerror = function() {

// note: xhr.readystate should be 4, and status should be 200. a status of 0 occurs //when the url becomes too large

if(xhr.status==0) {

alert('There was an error making the request. xmlhttprequest status is 0.');

}

else if(xhr.readyState!=4) {

alert('There was an error making the request. xmlhttprequest readystate is not 4.');

}

else {

alert('Woops, there was an error making the request.');

}

//console.log(xhr);

estimateLadda.stop();

selectLadda.stop();

};

xhr.send(jsonstring);

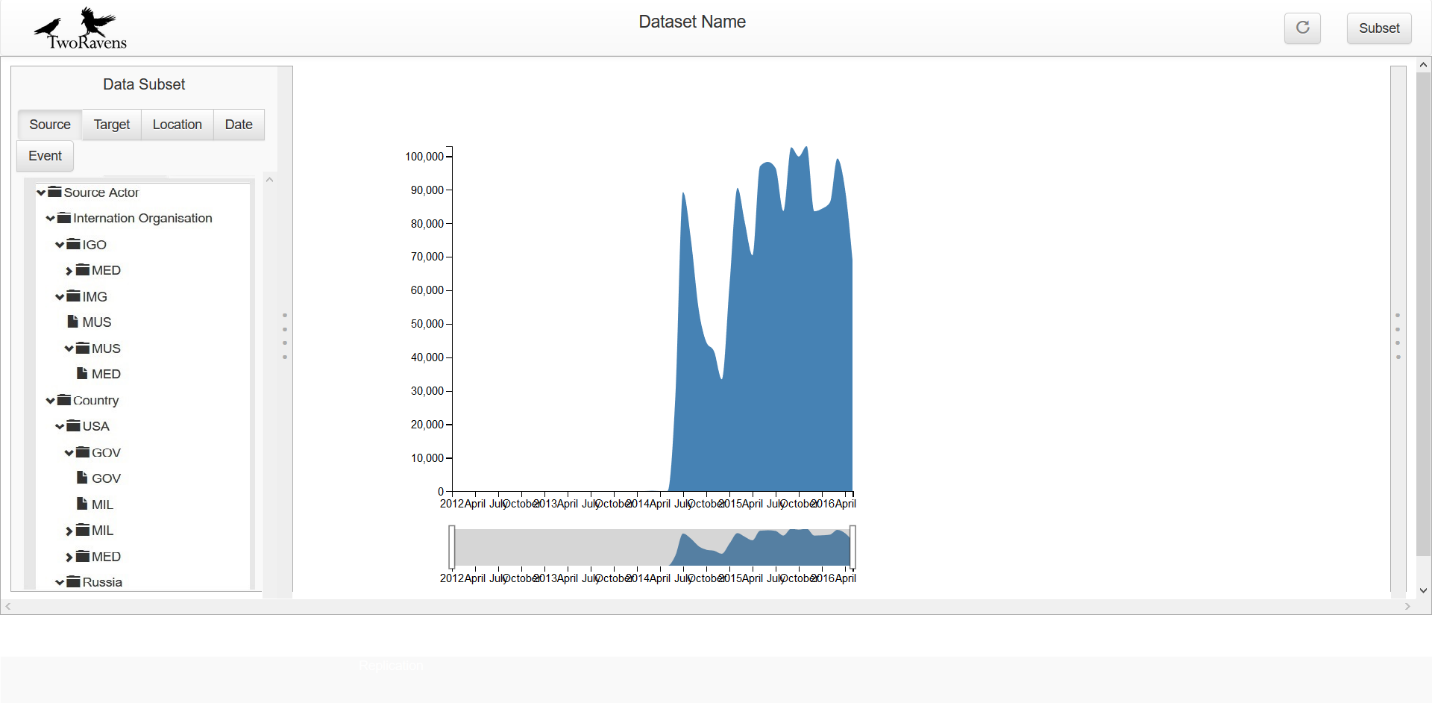
}//end of function

* Once the CORS request is sent, the query goes to the R server(rookquery.R), where it appends the query part of the JSON into the REST url, and calls the mongoDB web service. Once it gets back data, it sends a success message to the front end, which sets up the aggregation part of the code. The subset function also saves the subset data as a csv file for the aggregation file to read, so that the MongoDB need not be queried again during aggregation and is faster.

**Proposed Design:**

We are right now focusing on the subset part of the code only, as we are trying to integrate the subset, aggregate and analysis of Event Data handling right into the existing TwoRavens design language. The overlay interface, while effective, was a big leap from the usual seamless experience of TwoRavens, and might have a jarring effect on the user.

The good news, however is that because the backend and the query building part of the subset is done, we only need to concentrate on the UI elements. Here are some sketches of how we think the UI of subset should look:



* In the above sketch, the source and target actors will have a directory type structure for selection of the source actor and agent, where the relevant agents are nested under an actor. Clicking on the actor name selects only the actor, while clicking on the arrow opens the agent inside the actor and lets the user choose the agents for an actor. The structure showed in the sketch is taken from <http://bl.ocks.org/thehogfather/0e48ec486abbd5be17d7> . You can go to this link and see how this works, and what you can change. In order to get the structure, the data may need to be changed, and stores in nested JSON elements. The link above has given a sample JSON for the design, which you would need to automate, and write at the backend after querying the Source and Target lists.
* The graph above shows the distribution of the number of events by month of the entire dataset. This data is being read from a csv file, which has been created before running the program. I made the choice of making the csv file before running the program is that this data might not change frequently, and csv file can be refreshed every day, and not every time the site is being loaded. However, refreshing the csv file is still manual work, as we have to query the database, make the distribution table using the Table function in R, and then sorting it according to the date, and creating the csv file. If this function is written, there might be a way to update the csv file on a daily or weekly basis when this is hosted on a server. You might need to look into that.
* The distribution graph is interactive, where the user can zoom into expanding from year to months to days, and the using the slider given below the graph to pan the selection. This is a chart known as brush and zoom. Check it out here: <https://bl.ocks.org/mbostock/34f08d5e11952a80609169b7917d4172>
* Dr. Vito has proposed more distribution graphs for showing distribution of events according to actors and location, but that is still left to be implemented.
* The graphs will give an idea to the user before he/she selects data.
* To get idea for graphs and charts in D3.js, go to <https://bl.ocks.org/> to check out the different graphs and charts implemented by other users. These will help to select a good design in sync with TwoRaven’s own design language.
* The user will also be given an option to download the data after he/she subsets it, or if the user wishes to download the complete dataset. This has just been proposed, and is yet to be implemented.
* As a lot of work is left on Subset, Dr. Vito will guide you for Aggregation once subset in implemented, as the backend for aggregation is also written, and documented, but the theory of the code is outside my scope of understanding.
* The aggregation part has been done on R by anh, where she uses timeseries aggregation functions to aggregate using various levels of granularity of the temporal unit (daily, weekly, monthly, yearly), and has written functions to show those aggregations as plots. The timeseries functions and theory is out of my scope, and anh or Dr. Vito will fill you up on that.

That covers most of the current and proposed implementation of subset and installation of TwoRavens. This guide should be enough to get you started on subset, and then implement the rest of eventdata TwoRavens.

Look into the resources given below for further tutorials, and references, and if need be, please email me at [rohit.bhattacharjee99@gmail.com](mailto:rohit.bhattacharjee99@gmail.com).

**Resources**:

**D3.js**:

* <http://alignedleft.com/tutorials/d3>
* <https://github.com/d3/d3/wiki/Tutorials>
* <https://d3js.org/>

**JQuery**:

* <https://jquery.com/>
* <https://www.w3schools.com/jquery/>
* <https://jqueryui.com/>

**R**

* <https://www.r-project.org/about.html>
* <https://www.tutorialspoint.com/r/>
* <http://tryr.codeschool.com/>

More on event data:

* <https://github.com/Sayeedsalam/spec-event-data-server>
* <https://github.com/openeventdata>