To demonstrate how our transformation code works, we have set up a sample SQLite database named "sample\_225\_database.db" to run the code against. To mimic a typical REDCap EAV setup, this database contains the tables sample\_redcap\_metadata and sample\_redcap\_data, which have the same structure as the redcap\_metadata and redcap\_data tables found on a REDCap system.

Comment: screenshot of sample\_redcap\_metadata/data tables

To run the demonstration code, you need to have the sqlite3 library installed. This is necessary to execute scripts against an SQLite database.

The script below is used to create, insert and query the sample\_redcap\_data table. The sample\_redcap\_data table is the heart of the implementation as it contains all the data associated with the project. For demonstration purposes, we have incorporated the below functionality for a single project with a project\_id of 225. But the implementation is designed to be extensible for a wide variety of databases that reside in REDCap. We have included 3 events for demonstration to illustrate how events are managed. The record column has a one-to-one mapping with the patient\_id column in the sample\_patient\_id table.

The function create\_table\_data creates the sample\_redcap\_data table, and create\_inserts\_data inserts data read from the text file sample\_redcap\_data.txt. The data in the text file is evaluated as a python expression as shown in the code below, where it is read in as a tuple. We are calling the function executemany to perform all the inserts at one go.

To ensure that all data was inserted successfully, we also execute the select query to verify data insertion.

Code block

The code snippet below is used to create, insert and select the data from the sample\_redcap\_metadata table. The table contains the form name each field is associated with along with the field\_type which is represented by the element\_type column. The element\_type column contains one of the following three string values: “select”, “radio”, and “checkbox”. The element\_enum column contains the descriptions associated with a corresponding enumerated value, with '\n' used as a delimiter. For example, an element\_value field containing “1, female \n 2, male” assocates the value “1” with the description “female”, and the value “2” with description “male”.

The sample\_redcap\_metadata table is mainly used for joining with the sample\_redcap\_data table to fetch the form names associated with every field. Although the element\_enum column contains descriptions for every enumerate value, the structure of the descriptions makes it very difficult to join and fetch descriptions. In order to overcome this issue, we have created the gt\_lookup\_sample\_data table to join the enumerated values to descriptions. Details about this table are present below.

The function create\_table\_metadata creates the sample\_redcap\_metadata table, and create\_inserts\_metadata inserts data read from the text file sample\_redcap\_data.txt.

Code block

The script below contains functions for the creation, insertion and selection of data for the gt\_lookup\_sample\_data table. This table is used to fetch the descriptions of the codes present in the sample\_redcap\_data table. The descriptions present in the sample\_redcap\_metadata table are broken down using '\n' as the delimiter such that each value has a an associated description.

Viewing data in such a manner is desired while querying data for answering scientific research questions. The descriptions present in this table are fetched in the attribute tables by joining on this table thus making queries very intuitive.

Code block

The script below is used to create the attribute tables by field names. The base table sample\_redcap\_data table is split into separate attribute tables such that an attribute table for every field name is created. The script is used to generate SQL queries that create the attribute tables.

The attribute tables are created for both checkbox and non-checkbox fields.

Because checkbox fields in REDCap can conceptually represent one-to-many relations, we make a distinction between checkbox and non-checkbox fields. For example, the drug\_form allows for multiple drugs to be selected as being taken by a single patient. In the EAV schema, this results in multiple rows. For each drug selected, a row containing the record field and drug value is created.

Comment: insert image of patient with multiple drugs in the EAV table

Sometimes users want to query data like this in a more compact manner. So, we also create a view that groups checkbox values together with a group\_concat function.

Comment: insert image of drugs grouped together

The script below reads data from two separate csv files where one of the csv files contains the data associated with only checkbox fields and the other contains data associated with only non-checkbox fields.

The script appends the keyword "checkbox\_table" for all checkbox fields to provide a clear distinction and the non-checkbox fields' tables start with the string "table". The SQL that is output from this script is to be executed on the preferred SQLite environment (SQLIte command prompt, SQLite Studio, DBVisualizer, etc.). On execution, all the desired attribute tables by field names are generated.

The main advantage of attribute tables (as opposed to views) is that they increase query performance rapidly, since they are small in nature and also play a pivotal role for creating the views based on form names. These form name views are explained in the subsequent sections.

Code block

The script below fetches the patient id's for every patient. Each patient id has a one-to-one mapping with the record column present in the sample\_redcap\_table and all the attribute tables. In order to fetch the patient id, we join on the table created below.

Code block

The script below initially use the project\_id used for the project. Once the input is read, the function create\_view\_sql outputs the SQL query to construct the views by form names. The script generates the queries only for non-checkbox fields (i.e., select, radio) and these fields are identified by reading in a text file named "Samp\_dict\_no\_cb.txt". This text file contains a dictionary of form names as keys and associated field names as values. Then, the contents of text file are evaluated as a python expression.

All the view’s required columns are stored in reqlist variable, which is constructed with a list comprehension, and then type casted into a string to ease query construction. The SQL query to create the views is generated on running the script. This results in an SQL query to create a view for each form, which is then executed on the SQLite environment to create the view.

Code block

The script below creates the views by form names for only checkbox fields. It resembles the previous script with the only distinction being the string "checkbox\_" prepended to the query string. The dictionary read in contains the form names as keys and the checkbox-field names as values. This results in an SQL query to create a view for each form, which is then executed on the SQLite environment to create the view.