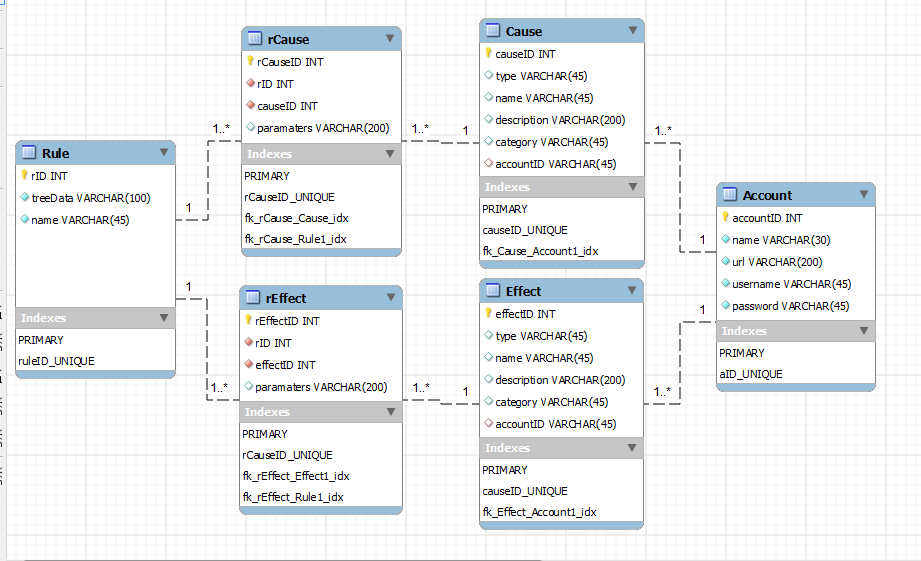
While developing the structure of the android application we decided that we needed a full database to hold all of the relevant information that we would need. This database would keep track of all causes, effects, rules, and account information so we needed it not only to be large but secure as well. We inquired with Samsung to see if they had any preferences on what we should use to accomplish this task and were presented with TouchDB (<https://github.com/couchbaselabs/TouchDB-Android>). We started playing around with it to try to get used to it but found the documentation for it to be subpar. After this setback we decided to go with SQLite since we needed to keep the database lightweight to keep our app from taking up too much memory on the phone as we remember it has to be constantly running in the background.

So getting started with this, we had to set up a structure for how we wanted the databases to look. This included how we would want our pulls to appear from the database. So we started with creating the objects that we would be using in our program to decide how many objects we would need to keep it lightweight while balancing that with how many variables were in each object. We also had to keep track of links between the objects to finally pull a full rule out of the database if we were to need that much information. I, personally, got to learn a lot about databases since I had never even been introduced to any sort of database language before this. There was a bit of a steep learning curve between not only learning SQL, but then to learn the limitations that were present with using SQLite as well as making the database so large with a number of tables and foreign keys.

Getting a little more specific into how the database was made, we have to first start by looking at how the objects were made that would basically be held by the tables. The cause object is one of the base objects so it needed to hold a lot of information including: cause id, type, name, description, category, and account id (which we will be implementing later). The cause id would of course be the primary key to keep track of which cause you would be talking about in the rCause table. It was at this time the autoincrement functionality became our friend as we discovered that it was already implemented in the tables and assigning a primary key actually renames the column to whatever you called it (led to many headaches and head scratching). The cause table would be kept separate from the rCause table since it needs to be an overarching list while the specifics that would needed to be implemented in each rule would be handled by the rCause table. The effect tables on the other side of the chart were basically mirror images of the cause tables.

When looking at the rules objects you realize that you need the string to keep track of how the rules interact (and/or) as well as a name to display and of course a primary key. A rule id is kept as well in the rCause/rEffect tables. Finally, the account table is not fully implemented yet in our application, but soon will be. The breakdown of the account (as we know right now) is that it must contain a (service) name, url, username, and password. All of these are essential in logging into certain other services such as Facebook, Twitter, etc. Below, you will see a display of how the tables talk to each other:



After that whole database is set up, you then need to be able to do multiple functions to accommodate for the database handler which is what the program needed to be able to get their information. Some of the most obvious ones were to insert causes and effects into the table. Whenever you start up the application for the first time you will be inserting all of the causes and effects from either a text file (to possibly be implemented later) or just being read in hard coded (current implementation). Now for inserting the rule you also need to be inserting the specific rCause and rEffect information since these are important parts of the rule. The application will be set to remember all of the rules from previous uses, because who really wants to have to input the rules every time they start up their phone.

Once you are done inserting all of the information for the application, there were the actual parts of the database that the other parts of the program needed to be able to access. The first ones I was told to implement were to return all cause, effect, and rule objects (separately). This was pretty easy as you just use an SQL statement to pull all of the information out of a certain table and put the information into an ArrayList. After those methods got done, we started to get a little more complicated in how we wanted to call from the database. The first method was to return all cause objects where the category of that cause is equal to a given category. So we passed a string in, used it in a SQL query to create a result table that you would pull from to get the wanted information. So of course the same thing also needed to be done for effects and rules. The rules one was actually more complicated since you had to go into the rCause and rEffect tables with external columns from other tables using joins (since there is no category column in the rules table or the rCause/rEffect).

So taking a break from the more complicated calls I had to return a full cause object given a cause id. After taking a break, it was back to complicated calls with receiving all rule objects where the type of that rule is equal to the given type, making sure that the effect list and cause tree are null, but the cause tree string is still there. This was for in the early parts of the application when you are pulling up some information quickly to display to the user while it doesn’t take up much processing power. And finally the last method I’ve had to make was to return an effect list given the rule id, which just made sure that we had all the method calls for the other parts of the program.