Hi guys, this SQL stuff is new and a little bit foreign. The recursive query stuff in particular can be tricky at first. So here's a quick guide on how I think about recursive queries. **Note, this is a supplement guide and recap of SQL. It is *not* a replacement for reviewing the lecture videos and slides. John's slides are extremely well done, so go look at them if you haven't already (especially the one on Recursive Select).**

## **First, a 5 minute refresher to SQL.**

SQL is a programming language for managing data. It is a form of declarative programming, which means you give the SQL interpreter information, and later ask it questions, without knowing exactly how SQL is providing the answers. This is yet another example of abstraction!

In SQL, you spend a lot of time making tables. You'll most often be using the create and select statements in order to build them. Here's an example using the SQL interpreter **(you may find it easier to read this code by copying and pasting it into sublime. Or hitting the left arrow button on the top left of Piazza to hide the nav bar):**

sqlite> create table cs61a\_instructors as  
 select "Andrew Huang" as name, "No" as retired, "Led Zeppelin" as favorite\_band, "potstickers" as favorite\_food, "Brian Harvey" as learned\_from union  
 select "Brian Harvey", "Yes", "The Beatles", "potstickers", "Hal Abelson" union  
 select "John DeNero", "No", "unknown", "potstickers", "Brian Harvey";

In this case, I made five columns: name, whether or not the instructor has retired, favorite band, favorite food, and who they learned 61A ideas from. Each select statement creates a new table, and union stacks them all together. create gives the resulting table a name that we can refer to later.

**Note: Any rumor that bringing John DeNero potstickers will get you an A in 61A is a lie.**

**It's important to memorize the following: select builds new tables either from scratch, or from previously created tables. This means it creates a new set of rows with particular columns. For example, in cs61a\_instructors table above, there were three select statements, each building one row. Each select generated a row with three columns.**

Using select, you can query for all of this information back:

sqlite> select \* from cs61a\_instructors  
Andrew Huang|No|Led Zeppelin|potstickers|Brian Harvey  
Brian Harvey|Yes|The Beatles|potstickers|Hal Abelson  
John DeNero|No|unknown|potstickers|Brian Harvey

Or you can query for just a specific column:

sqlite> select name from cs61a\_instructors;  
Andrew Huang  
Brian Harvey  
John DeNero

You can also filter your query for important information using the where clause.

sqlite> select name from cs61a\_instructors where favorite\_band = "The Beatles";  
Brian Harvey

Lastly, you can concatinate strings together for the lulz:

sqlite> select name || " teaches CS 61A" from cs61a\_instructors where retired="No";  
Andrew Huang teaches CS 61A  
John DeNero teaches CS 61A  
sqlite> select name || " eats " || favorite\_food from cs61a\_instructors;  
Andrew Huang eats potstickers  
Brian Harvey eats potstickers  
John DeNero eats potstickers

## **Joins**

Now imagine we had another table:

sqlite> create table bands as  
 select "The Beatles" as name, 1960 as founded, "Rock, Pop" as genre, "Helen Shapiro" as opened\_for union  
 select "Led Zeppelin", 1968, "Hard Rock", "Vanilla Fudge";

Let's say we wanted to find out which which music genres the instructors listen to. We would have to do something called a join on these two tables. We use the from clause to give us access to the columns of both tables, and we look up rows that match the given conditions of the where clause.

sqlite> select b.genre, i.name from bands as b, cs61a\_instructors as i where i.favorite\_band = b.name  
Hard Rock|Andrew Huang  
Rock, Pop|Brian Harvey

Here, we created a new table with new rows based on joining the different rows of the two tables together. The above select statement reads "Create new rows which have two columns, the genre column from bands, and the name column from cs61a\_instructors. Make sure the new rows are created by matching together rows from the two tables, such that the name of the band (in the bands table) matches the favorite band (in the cs61a\_instructors table)."

**This matching business is really important to be able to picture in your head. It makes the recursive query stuff make a lot more sense. Thus I'm going to spell out the results in the following blockquote of how I think about it. Feel free to skip the blockquote, but come back if you get stuck.**

Let's consider where Hard Rock|Andrew came from in the results. The SQL interpreter hunts around and finds rows in bands whose data in the "names" column match up with the data in the "favorite\_band" column among the rows for cs61a\_instructors. It first finds a match with the row Led Zeppelin|1968|Hard Rock in bands and the row Andrew Huang|No|Led Zeppelin|potstickers|Brian Harvey in cs61a\_instructors. With those two rows selected, it takes the appropriate information from the columns of both tables, namely the genre of the band (Hard Rock) and the name of the instructor (Andrew Huang), and creates the new row, Hard Rock|Andrew. The SQL interpreter then repeats this for all other possible matches.

Note that we also had to alias the two tables, since both tables had a name column. Otherwise, SQL doesn't know which column 'name' that we're talking about:

sqlite> select name, name from bands, cs61a\_instructors where favorite band = name;  
Error: ambiguous column name: name

The neat thing is that we can join a table with itself. For example, we can get a list of all compatible instructors who like the same food:

sqlite> select a.name, b.name from cs61a\_instructors as a, cs61a\_instructors as b where a.favorite\_food = b.favorite\_food and a.name != b.name;  
Andrew Huang|Brian Harvey  
Andrew Huang|John DeNero  
Brian Harvey|Andrew Huang  
Brian Harvey|John DeNero  
John DeNero|Andrew Huang  
John DeNero|Brian Harvey

Why did we say a.name != b.name? Could we have reduced the duplicates?

## **With and Local Tables**

I can build one-off tables that hold local information (as opposed to using create, which would give a global name to a table) by using the withconstruct:

sqlite> with  
 m(mentor, mentee) as (  
 select learned\_from, name from cs61a\_instructors  
 )  
select \* from m;

Here I made a quick table of mentors and mentees. The syntax indicates that the name of this local table is "m", and that it has exactly two columns, "mentor" and "mentee". Following that is a select statement that describes such a table. Afterward, we can do a one time select from that table in order to list out all the mentor/mentee pairings. (Yes, this example is kind of contrived.)

**Sidenote#1:** If we wanted to make this table permanent, what would we have to do?

**Sidenote#2:** Is there a way to list only the mentors that don't appear as rows in cs61a\_instructors? (i.e. how do we get Hal Abelson?)

(Answers to these are important and are discussed below, but think about them yourself first!)

**It's really important to realize that these with constructs aren't function calls. You're thinking of a Different Language.**

Here's another one. What does this one do?

sqlite> with  
 my\_friends(name, a\_lil\_help) as (  
 select name, opened\_for from bands  
 )  
select name || " owes a lot to " || a\_lil\_help from my\_friends;

## **Recursive Queries**

H'okay, so we finally reviewed enough SQL to start talking about recursive queries. Let's say we wanted to build lists of linages of 61A instructors. That is, we'll list out who learned from who like the following:

Hal Abelson  
Hal Abelson>Brian Harvey  
Hal Abelson>Brian Harvey>Andrew Huang  
Hal Abelson>Brian Harvey>John DeNero

It turns out, it's easier to think about this problem if we have a table that describes just the mentor mentee pairings:

sqlite> create table mentors as  
 select "Hal Abelson" as mentor, "Hal Abelson" as mentee union  
 select learned\_from, name from cs61a\_instructors;

To answer Sidenote#1, we can use the create statement to give a permanent name to the table of mentors.

To answer Sidenote#2, given our current tables and what we know of SQL, there isn't really an easy way to get "Hal Abelson", so we had to hardcode him in here as his own mentor. This makes the following possible:

sqlite> with  
 linage(line, predecessor) as (  
 select mentor, mentee from mentors where mentor = mentee union  
 select line || ">" || mentee, mentee from mentors, linage where predecessor = mentor and mentee != mentor  
 )  
select line from linage;

So in order to create the lineages we care about, we need to keep track of the current line (a string of names separated by ">"), as well as the last person we added to the line. Thus our local table linage has two columns, the line column and the predecessor column.

Note the first select statement. It starts our table off with the row Hal Abelson, Hal Abelson because we note that if you taught yourself, then you are the root of all linages (everyone else must learn from you).

We then union this with the recursive select statement. By saying from mentors, linage, we get access to the mentor and mentee columns from the mentors table, as well as the line, predecessor columns from the linage table itself.

From there, we know that the new rows we want to generate are the result of taking an existing line and adding one more person to the end.

We specify that that person must be a mentee of the predecessor (where predecessor = mentor).

Why is this recursive? The key observation is that the in order to generate linage, we refer to the columns from linage itself!

**Here is a blockquote going into more detail. Again feel free to skip it, but remember to come back if you get stuck somewhere else:**

We start by imagining the linage table. It has two columns, line and predecessor, but at first, there are no rows. Then, we add our first rows via the first select statement. That select statement pulls from the mentors table and creates the row Hal Abelson|Hal Abelson. Thus now our linage table has exactly one row in it.

Now the other 3 rows (remember the results from above) are all generated by repeatedly executing the second select statement. Remember that the SQL interpreter creates as many rows as it can whenever it interprets a select statement. That can result in zero, one, or many rows. So when the second select statement refers to the liange table itself, we add one row at a time, each building from the ones before it.

For example, Hal Abelson>Brian Harvey|Brian Harvey row is generated with the assumption that Hal Abelson|Hal Abelson is already in the linage table. The SQL interpreter matched Hal Abelson|Hal Abelson row in the linage table to the Hal Abelson|Brian Harvey row in the mentors table. From there, the interpreter took the data it needed from those two rows, the line (Hal Abelson), and mentee (Brian Harvey) to construct the new row, Hal Abelson>Brian Harvey|Brian Harvey. The interpreter continues to generate Hal Abelson>Brian Harvey>Andrew Huang|Andrew Huang, and then Hal Abelson>Brian Harvey>John DeNero|John DeNero in the same manner.

Lastly, the select statement that is outside the with clause selects only the line column from the linage table, so we skip on the predecessors in our results.

**Sidenote#3:** Why is mentee != mentor in the recursive query as well? What row causes this to be necessary? What happens if we take it out?

So the general plan of attack for writing these is

1. Understand what the question is asking
2. Figure out a reasonable base case that generates the starting rows. This is generally some select statement that does not make a recursive query.
3. Given that the rows from our base case query, come up with a relationship between columns that allows us to generate the rest of the rows we care about. Note that we always need to generate as many columns as our table specified. In our example above, we needed to generate a new line and a new predecessor .

Hope this helped. Post comments or questions in followups.

- Andrew

P.S. Also, all the examples are tested, so here are the tables so that you can copy and paste them and test the above examples out for yourself (**recommended!**):

create table cs61a\_instructors as  
 select "Andrew Huang" as name, "No" as retired, "Led Zeppelin" as favorite\_band, "potstickers" as favorite\_food, "Brian Harvey" as learned\_from union  
 select "Brian Harvey", "Yes", "The Beatles", "potstickers", "Hal Abelson" union  
 select "John DeNero", "No", "unknown", "potstickers", "Brian Harvey";  
  
create table bands as  
 select "The Beatles" as name, 1960 as founded, "Rock, Pop" as genre, "Helen Shapiro" as opened\_for union  
 select "Led Zeppelin", 1968, "Hard Rock", "Vanilla Fudge";  
  
create table mentors as  
 select "Hal Abelson" as mentor, "Hal Abelson" as mentee union  
 select learned\_from, name from cs61a\_instructors;