SQL and Way Too Many ManyToMany Relationships

<p><b>Filtering Things Out From Queries</b><br>

One interesting problem that Mark and I faced when working on our swim workout generating website (<a href="/blog/view/poctoproduct">blog post</a>) was that of filtering out swim sets that a user wouldn't want. We realized that while many users would love the workout variety our web app was capable of producing, a top desired feature would be to filter out, for example, "all breaststroke sets" (my least favorite stroke). </p>

<p><b>Document Based Database?</b><br>When I first modeled this capability, Mark and I were using a document form of a "database" to crudely test the swimscript (this was way before we decided on a website over an app). In the document, each bucket was a variable that contained a list. The list contained numerous dictionaries. Each dictionary had similar properties: set\_description, intervals, and eventually "tags". We tagged each set as either being a "fly set" or a "kick set" or a "hold the breath set". All of these were potentially tags that we would want filtered out when our script went to go get the right kind of swim set for the user.</p>

<p> <b> Just One Query of All Possible Worlds</b><br>

When I looked into how we should model this capability with our MySQL tables and django models, it came apparent that this was a classic ManyToManyRelationship: <em>Many</em> swim sets could be labeled with one tag such as "butterfly", and one swim set could be labeled with <em>many</em> different tags. A single field in our swimset table "tag\_id" wouldn't work (ie a foreignkey). However, when I implemented things in the "standard way" by having a relation table called swimset\_styletag, I discovered a major problem. For every tag that a single swimset would have, it would be listed <em>again</em>, multiple times, in the swimset\_styletag table for each and every style\_tag. </p>

<div class="code">

SELECT swimset.id FROM swimset\_styletag <br>

… INNER JOINS… <br>

WHERE swimset\_id NOT IN (SELECT \* from swimset\_styletag WHERE styletag\_id IN (1,2,3));<br>

</div>

<p>(note: for simplicity sake of this section I will pretend that my SQL is selecting a specific set rather than a random one)</p>

<p>

A query like that only seemed to filter out the exact queried table of NOT IN (style tag 1,2,3). If we were trying to filter out sets tagged any combination of 1 and/or 2 and/or 3, we would need to NOT IN filter multiple combinations of: NOT IN (1,3), NOT IN (1) NOT IN (1,2) NOT IN (2), NOT IN (3) etc. This is because we would want to filter out a set, no matter what, even if it had just one tag from the list of tags 1, 2 or 3. Because we have 20+ tags, this standard ManyToMany system just wouldn't scale:

<div class="code">swimset\_styletag<br>

<br>

id swimset\_id styletag\_id<br>

-- ---------- -----------<br>

1 &nbsp; &nbsp; &nbsp; 1 &nbsp; &nbsp; &nbsp; 1 <br>

2 &nbsp; &nbsp; &nbsp; 2 &nbsp; &nbsp; &nbsp; 2 <br>

3 &nbsp; &nbsp; &nbsp; 3 &nbsp; &nbsp; &nbsp; 1 <br>

4 &nbsp; &nbsp; &nbsp; 3 &nbsp; &nbsp; &nbsp; 2 <br>

<br>

swimset<br>

<br>

id set\_description <br>

-- -----------------------<br>

1 &nbsp; 4 x 100 Free Interval A<br>

2 &nbsp; 4 x 100 Back Interval A<br>

3 &nbsp; 6 x 100 free/back on in<br>

<br>

styletag<br>

<br>

id tag\_name <br>

-- ---------- <br>

1 &nbsp; Free <br>

2 &nbsp; back <br>

</div>

<p> <b>The Boolean Hack</b><br>

So instead I did something that SQL database administrators are going to cringe over. I simply made every tag into a true/false value on the one swimset table.</p>

<div class= "code">

swimset<br>

<br>

id set\_description &nbsp; &nbsp; &nbsp; &nbsp; free\_tag back\_tag<br>

-- ----------------------- -------- --------<br>

1 &nbsp; 4 x 100 Free Interval A &nbsp; True &nbsp; &nbsp; False<br>

2 &nbsp; 4 x 100 Back Interval A &nbsp; False &nbsp; &nbsp;True<br>

3 &nbsp; 6 x 100 free/back on in &nbsp; True &nbsp; &nbsp; True<br>

</div>

<p> This means that I can now easily filter out "freestyle sets", by querying for sets that have "free\_tag = false". The advantage to earlier is that I don't have to account for every possible combination of all the other tag's Boolean values. I merely need to specify which tags I definitely do not want.

<div class="code">

SELECT \* <br>

FROM swimset <br>

WHERE free\_tag = 0 AND back\_tag = 1; <br>

</div>

<p> However, there was still the problem of needing "every" combination of query based on user inputs. If we have 20 tags, that means we could have up to 20 x 19 x 18 x 17 … x 2 x 1 combinations of SQL queries! That's way too many for "if … else if" statements to handle. So here's what I came up with to solve this problem:</p>

<div class="code">

cur.execute( <br>

<br>

"SELECT \* <br>

FROM swimset <br>

WHERE (free\_tag = 0 OR %s) AND (back\_tag = 0 OR %s);", <br>

<br>

user\_input\_list)<br>

</div>

<p>When users choose to filter out nothing, their tag inputs are set to "tag\_XYZ = true". These are then passed into the python SQL (cur.execute above), and the SQL evaluates as: </p>

<div class="code">

free\_tag = 0 OR 1<br>

</div>

OR

<div class="code">

free\_tag = 0 OR 0 <br>

</div>

<p> "0 OR 1" evaluates to "true" and no filtering out of that tag occurs. "0 OR 0" evaluates to "false" and those sets tagged as such are filtered out. So while I did break some sort of database design rules by putting these ManyToMany relationships all into one table as true/false, I did uphold the ultimate programming rule of Don't Repeat Yourself. I avoided 20 x 19 x 18 x 17 … x 2 x 1 control flow statements. </p>