Breaking Up with Lists - When you are Ready for a Real Relationship

Posted At : December 7, 2005 11:55 AM | Posted By : Mark Kruger
Related Categories: Coldfusion & Databases

Every blue moon someone asks, "How do I determine if a list contains a value using SQL?" This question belies a misunderstanding of how a relational database is supposed to work. You *can* get at a value in a list using a UDF (see this **previous post**). But just because something *can* be done does not mean that it *should* be done.

Why Use Lists

In your application Code lists are great. That doesn't mean they are appropriate in every case. In fact you should avoid lists in favor of structures or arrays whenever the data you are manipulating is variable character data (where the delimiter could be in play) or the size of the data pool is very large. Using a list of 10 or 12 (or even 50) static hex values might be ok, but using it to track all the possible hex values in a color wheel is probably *not* a good idea. Using it for a list of presidents names will work fine, but using it for the entire genealogy of FDR back to Adam is a bad idea.

In the database, however, lists are not so good. A database doesn't have all those nice easy to grasp functions like "listfind()", "listgetat()" and "listfirst()" to work with. Sure, you can create those functions using SQL UDF's, but when you do you are pouring ketchup on a filet mignon. You see, when it comes to lists your database is *ready for a relationship*. Now those of you with commitment anxiety take a deep breathe - we are talking about database design here. Suppress your flight instinct or pretend you are in Vegas and stay with me.

Databases and Relationships

To a database a list is really a relationship to a separate set of data. You can easily handle the *idea* of a list using database relationships - and it's not that hard. The first task that awaits you is to figure out what your list represents. Let's take an example that's pretty easy to grasp - user and group permissions. In fact, I've seen a few examples of code that use a list of Ids or strings to indicate the "groups" to which a user belongs. Let's say you are user "sam" and you have the following list: "Admin,Reports,Manager,Maintenance". This list is stored in the database in a column called "groups". Before you can use a certain application (let's say a report) the application code checks to see if you are in the "reports" group by doing something like:

```
<cfif listfindnocase(userQuery.groups,"reports")>
      .... access to a link or application....
   </cfif>
```

That works just dandy - what's wrong with it? Well, for one thing, the group permissions are *only accessible at the application level*. They are not accessible at the database level - at least not without a bunch of SQL gymnastics. For example, what if you had a manager application where the data displayed depended on the group to which the user belonged? You would either have to "pre-process" the groups and pass them into your query as strings, or you would have to do something with an SQL UDF. But if you were using relational tables you would be able to do joins or sub queries or views that leveraged the power of SQL and provided results back to the application code that were pre-filtered.

Onother problem with this approach is that it is "one way". I have to start with the user and check the group. What happens when I want to start with the group and see a list of users? My options become pretty limited. I could do the UDF work-around, or I might select *all* the users and filter them out when looping through them by checking the list, but neither option is ideal.

Playing Dr. Phil

How many psychologists does it take to change a list? Only 1 - but the list has to really *want* to change. Taking a list based approach and moving it to a database schema takes some thought. The first thing to determine is the type of relationships you wish to support. There are 3 main types, one to one, one to many and many to many. The main question is, are either the users or groups exclusive. In other words, can any number of users belong to any number of groups. If the answer is "yes" then you are dealing with a "many to many" relationship.

Matchmaking

Now that I know the that I'm working with a "many to many" situation, what's next? Let's finish off our example. First, let's assume you have a user_id in the user table as the primary key - and a "group_id" as the primary key in the group table. What is needed a third table, a cross reference table (a so-called "bridge" table), that contains rows of user id's and group ids let's call it "cfx_usergroup". You can then query all sorts of group permissions by joining or manipulating the 3 tables together. For example.

```
<cfquery name="checkAdmin" datasource="#dsn#">
   SELECT    U.user_id, u.userName, g.group_id, g.groupName
   FROM   user U JOIN cfx_usergroup cf
   ON      U.user_id = cf.user_id
   JOIN   groups g
   ON      g.group_id = cf.group_id
   WHERE   u.user_id = 484
   AND      g.GroupName = 'Admin'
</cfquery>
```

But why? Why would you want this code when you could have just "listfind(groupList,'Admin')"? Because it's extensible. You can now tie other types of things in the database schema to permissions. You can create views of data that are filtered by user based on which groups to which they belong. You can add groups and entities more easily without reengineering your code. You can even create downstream behavior without altering user permissions. Your design can also more easily be represented as a database schema with the relationships.

Some Caveats

I know I know ... you inherited the list system and now you have to work with it. Sometimes this is the case. The SQL UDF approach can be helpful in this case, but clearly it is a work around. It is *not good database design* to store things in lists in character columns that are going to be treated like individual data bits and filtered accordingly. Where your application calls for a database relationship you should use one. If the existing database is wrong, then you should recommend as the first option that it be changed, and you should make it clear that work arounds save money *only in the short term*. In the long run they always generate more work arounds.