Manager.cfc and Elasticity

Posted At : November 30, 2010 10:30 AM | Posted By : Steve
Related Categories: CF_DMQuery, com.sebtools, DataMgr

Whenever I see any example saying "See how easy this is!?", I inevitably think "What happens when my needs differs a bit from the expected?". My worry is that I could start with a situation for which the tool is a good match and then things change a bit. Do I have to abandon the tool altogether in those cases? Basically, I always want to know how much elasticity any solution has.

More often than not, it seems, systems that are extremely efficient have very little elasticity (think of airline schedules where bad weather anywhere effects flight patterns everywhere). I think that the past several blog entries on **Super-Easy CRUD/File Management** have demonstrated that the **com.sebtools package** provides great efficiency for basic CRUD and file tasks. What it hasn't demonstrated (yet) is how elastic it is when the problem is more complicated than what com.sebtools is built to handle.

In the **last entry**, I showed how **Manager.cfc** can easily filter by equality for database fields or for **relation fields** and can nearly as easily filter for inequality on both as well. Relation fields themselves provide a lot of flexibility in terms of what data you can retrieve and filter by without writing any SQL.

Relation Fields with Filters

The built-in relation fields cover most situations, but sometimes we want a subset of the data provided by one of them. For example, we automatically get a count of the number of employees in each department. We might, however, want the number of employees making more than $250,000 in any department. To do this, we could create a **relation field with a filter**.

```
<field name="NumHighPaidEmployees" Label="Highly Paid Employees">
   <relation
      type="count"
      entity="Employee"
      field="EmployeeID"
      join-field="DepartmentID"
   >
      <filter field="Salary" operator="GTE" value="250000" />
   </relation>
</field>
```

We could even have multiple filters in any relation field. For example, we could have a field with the number of employees in a department who are paid more than $250,000 and hired since 2009.

```
<field name="NumHighPaidEmployees" Label="Highly Paid Employees">
   <relation
      type="count"
      entity="Employee"
      field="EmployeeID"
      join-field="DepartmentID"
   >
      <filter field="Salary" operator="GTE" value="250000" />
      <filter field="HireDate" operator="GTE" value="2009-01-01" />
   </relation>
</field>
```

Custom Relation Fields

Sometimes, however, we need a field that is more complicated than any of the built-in relation fields. In that case, we can use **custom relation fields**. So, for example, if we wanted to know if the first and last name fields for an employee were the same then we could define that relation field like this:

```
<field name="areNamesSame">
   <relation
      type="custom"
      CF_Datatype="CF_SQL_BOOLEAN"
      sql="
         CASE
         WHEN FirstName = LastName
         THEN #variables.DataMgr.getBooleanSqlValue(true)#
         ELSE #variables.DataMgr.getBooleanSqlValue(false)#
         END
      "
   />
</field>
```

The "sql" attribute of the "relation" element is the SQL that should execute for this relation field. The "CF_Datatype" attribute is required if you want to be able to filter by the field as well (otherwise **DataMgr** won't know what SQL type to use for the comparison). In this example, I used the "getBooleanSqlValue" method which returns the SQL for a boolean value for the database engine being used. This is because different database engine use different values for boolean fields. For example, MS Access can accept 1 or 0 or "yes" or "no" whereas SQL Server will only accept 1 or 0 and PostGreSQL will accept true or false (which SQL Server won't), but not 1 or 0.

Insert Custom SQL

Getting even more complicated, we may sometimes want to **inject custom SQL** into a query. We can do this with the "AdvSQL" argument.

For example, we might want to get a count of employees by DepartmentID. This could be written as this SQL:

```
SELECT      CategoryID, count(EmployeeID) AS NumEmployees
FROM      hrEmployees
GROUP BY   CategoryID
```

If, however, we want to continue to take advantage of the other filters that we have built in already so that (for example) we could get only the employee hired in a certain year who make over $250,000 and work in a certain department and were hired after a certain date then we might want to just inject a bit of SQL into the query:

```
<cfset sAdvSQL = StructNew()>
<cfset sAdvSQL["SELECT"] = "count(EmployeeID) AS NumEmployees">
<cfset sAdvSQL["GROUP BY"] = "CategoryID">
<cfset arguments["AdvSQL"] = sAdvSQL>
<cfset getRecords(argumentCollection=arguments)>
```

Then again, we might just get more mileage by using an **aggregate relation field** from the Categories table.

We might also want to add an argument to our "getEmployees" method called "FirstOrLastName" which would filter employees by either the first name or last name (not something that a relation field could handle). Here would be our Employees.cfc:

```
<cfcomponent extends="com.sebtools.Records">

<cffunction name="getEmployees" access="public" returntype="query" output="no">

   <cfif StructKeyExists(arguments,"FirstOrLastName") AND Len(arguments.FirstOrLastName)>
      <cfset arguments.AdvSQL = StructNew()>
      <cfset arguments.AdvSQL.WHERE = "
         (
               FirstName = '#arguments.FirstOrLastName#'
            OR   LastName = '#arguments.FirstOrLastName#'
         )
      ">
   </cfif>

   <cfreturn getRecords(argumentCollection=arguments)>
</cffunction>
</cfcomponent>
```

This syntax, however, gives up the protection of cfqueryparam. Here is the syntax with that protection:

```
<cfcomponent extends="com.sebtools.Records">

<cffunction name="getEmployees" access="public" returntype="query" output="no">

   <cfif StructKeyExists(arguments,"FirstOrLastName") AND Len(arguments.FirstOrLastName)>
      <cfset arguments.AdvSQL = StructNew()>
      <cfset arguments.AdvSQL.WHERE = [
         "(",
         "      FirstName = ",
               variables.DataMgr.queryparam(cfsqltype="CF_SQL_VARCHAR",value=arguments.FirstOrLastName),
         "   OR   LastName = ",
               variables.DataMgr.queryparam(cfsqltype="CF_SQL_VARCHAR",value=arguments.FirstOrLastName),
         ")",
      ]>
   </cfif>

   <cfreturn getRecords(argumentCollection=arguments)>
</cffunction>
</cfcomponent>
```

Custom Query

Sometimes, however, we want to write a query of suffient complexity that Manager (and even DataMgr) just get in the way. It is usually possible to do this with AdvSQL, but harder than writing the query from scratch. The only challenge, however, is that you may still want to take advantage of relation fields that you have already defined or the built in tracking of logically deleted records.

Fortunately, we can get the best of both worlds using the **CF_DMQuery** tag. This allows you to write your own query and inject DataMgr sql into it (sort of AdvSQL in reverse).

For example, maybe we would like to find out the number of employees hired in each year. Here would be the manual SQL:

```
SELECT      count(EmployeeID), Year(DateHired) AS YearHired
FROM      hrEmployees
GROUP BY   Year(DateHired)
```

In order to remove any logically deleted records and filter out records by any in incoming arguments, you could just do this:

```
<cf_DMQuery name="qEmployees">
SELECT      count(EmployeeID), Year(DateHired) AS YearHired
FROM      hrEmployees
WHERE      1 = 1
         <cf_DMSQL method="getWhereSQL" tablename="hrEmployees" data="#arguments#">
GROUP BY   Year(DateHired)
</cf_DMQuery>
```

You could also have a manual query where you want to manually include a single relation field. This query won't be complicated enough to justify the use of cf_DMQuery, but will show that functionality:

```
<cf_DMQuery name="qEmployees">
SELECT      EmployeeID,
         <cf_DMSQL method="getFieldSelectSQL" tablename="hrEmployees" field="Category">
FROM      hrEmployees
WHERE      1 = 1
         <cf_DMSQL method="getWhereSQL" tablename="hrEmployees" data="#arguments#">
</cf_DMQuery>
```

You might also want to do a join by another table. For example, let's say we want to do an outer join with an "hrRegions" table to get the name of their region, if they are in one. We could make our "getEmployees" method as follows:

```
<cffunction name="getEmployees" access="public" returntype="query" output="no">

   <cfset var qEmployees = 0>

   <cfif NOT StructKeyExists(arguments,"fieldlist")>
      <cfset arguments.fieldlist = "">
   </cfif>

   <cf_DMQuery name="qEmployees">
   SELECT      <cfif ListFindNoCase(arguments.fieldlist,"RegionName") OR Len(arguments.fieldlist) EQ 0>
            hrRegions.RegionName,
            </cfif>
            <cf_DMSQL method="getSelectSQL" tablename="hrEmployees" fieldlist="#arguments.fieldlist#">
   FROM      hrEmployees
   LEFT JOIN   hrRegions
      ON      hrEmployees.RegionID = hrRegions.RegionID
   WHERE      1 = 1
            <cf_DMSQL method="getWhereSQL" tablename="hrEmployees" data="#arguments#">
   </cf_DMQuery>

   <cfreturn qEmployees>
</cffunction>
```

In point of fact, this doesn't hold any advantages over setting up a relationship with the hrRegions table. The efficacy of this particular example is less significant, however, than the capability itself.

It is important to note that this takes connects to DataMgr directly, skipping over Manager.cfc and giving up the "#name#URL" and "#name#Path" columns that Manager.cfc creates for any field fields (as discussed in "**Super-Easy CRUD/File Management**").

In order to get those back, use the following code:

Note, however, that the file name field will have to be present in the query for Manager.cfc to add the "#name#URL" and "#name#Path" columns to the query.

Conclusion

As you can see, Manager.cfc and DataMgr.cfc (the relevant pieces of com.sebtools for this discussion) offer plenty of elasticity to go with their power. So, even if your situation grows more complicated than what they were built for, you can still take advantage of them in your application and not lose the value of the work that you have already done.

The **com.sebtools package** is open source and free for any use.