How does the muaggatt table get populated?

This table exists in the SDM database (also SDMOffline and SDMOnline).

I started looking into the source code of many projects and the Dev databases. For the Dev databases, I had to search through all stored procedures, functions and tables for clues. I found that:

the SDMOffline database has a stored procedure called "SDMDM\_InsertMuaggatt",

the sdwstaging database has SP "StagingExport\_muaggatt",

the nasis database has SP “NasisToSSURGO\_muaggatt”, and

the SDMMgtOnline database has tables "aggtabcol", "aggtabcolconstraint", "DataManagerTask", "DataManagerTaskRequest", and "DataManagerBlockedTask".  
  
In SDMOffline, there is the Stored Procedure (SP), SDMDM\_InsertMuaggatt:

PROCEDURE [dbo].[**SDMDM\_InsertMuaggatt**] @DataTable [dbo].[MuAggAttTable]

AS

INSERT INTO [dbo].[muaggatt]

([musym],[muname], ... ,[awmmfpwwta],[mukey])

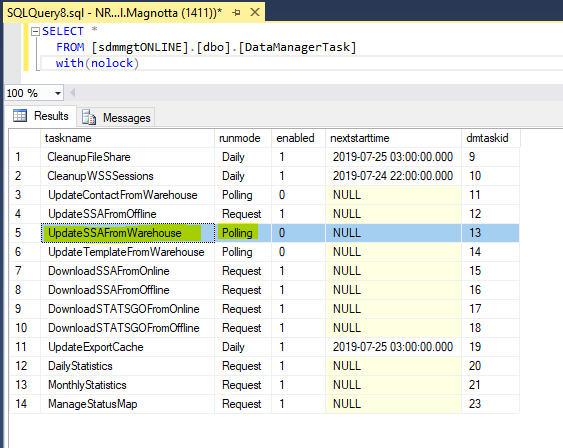
SELECT [musym],[muname], ... [awmmfpwwta],[mukey]

FROM @DataTable

This SP is used to populate the muaggatt table from a "data table" ADO object.

I performed a search for the string “SDMDM\_InsertMuaggatt” across all my applications’ source code files. I only found it in the SoilDataManagerService solution at “C:\SoilDataManagerService\_Folder\trunk\WorkItems\UpdateSSAFromWarehouse.cs”. The SoilDataManagerService is a Windows service and exists in the Soils Dashboard. There is a “tab” in the Soils Dashboard GUI (Graphical User Interface) labeled as “Data Manager”.

This SP, “SDMDM\_InsertMuaggatt”, is invoked by the SoilDataManagerService, run as a task under the Soils Dashboard.

In the Soil Data Manager solution, C:\SoilDataManagerService\_Folder\trunk\ SoilDataManagerService.sln, is the invocation of the SP in file “C:\SoilDataManagerService\_Folder\trunk\WorkItems\**UpdateSSAFromWarehouse**.cs”.  
  
If we look at the sdmmgtOnline database, there is a table called “DataManagerTask”. The tasks for the Data Manager are shown below. Notice the “UpdateSSAFromWarehouse” task.  


Below I show how the code runs. In file UpdateSSAFromWarehouse.cs, we start with the **Run() method which calls method TransferArea()**.

public override void **Run**(object threadContext)

{

LogDebug("Starting task execution");

RunStatus = "Polling for updated survey areas in data warehouse.";

try

{  
**TransferArea**(); //Run() calls TransferArea()  
…

In file UpdateSSAFromWarehouse.cs, method **TransferArea() calls LoadMyaggatt()**. What we are doing is updating the Soil Survey Area from the Warehouse (sdw database) to the SDMOffline database.

Beginning at line 193 in method TransferArea() is the following code:  
if (\_operation == SdmTransferOperation.Add || \_operation == SdmTransferOperation.Update)  
{  
try

{

**LoadMuaggatt**(); //TransferArea() calls LoadMuaggatt()  
}

…  
  
File “UpdateSSAFromWarehouse.cs” has a method called **private void LoadMuaggatt()** from lines 2596 through 2655.

The code in LoadMuaggatt() starts out by creating a skeleton for the muaggatt table. That is, all the fields of the muaggatt table are selected and joined with the mapunit table based upon the mukey field or “map unit key”. Below is the abbreviated code in the method, and then the actual SQL SELECT statement and some output in SSMS.

// Create skeleton for muaggatt table with only musym and muname populated (mukey also)

SELECT

mapunit.musym as musym,

mapunit.muname,

null AS mustatus,

muaggatt.slopegraddcp,

...

...

muaggatt.awmmfpwwta,

mapunit.mukey

FROM

mapunit temp\_mapunit

INNER JOIN mapunit ON temp\_mapunit.mukey = mapunit.mukey

LEFT JOIN muaggatt ON muaggatt.mukey is null

This is the SQL text placed in an SSMS query and executed.

--musym, muname and mukey will be populated, the rest will have NULL values

USE sdmOFFLINE;

SELECT

mapunit.musym as musym,

mapunit.muname,

mapunit.mukey,

null AS mustatus,

muaggatt.slopegraddcp,

muaggatt.slopegradwta,

muaggatt.brockdepmin,

muaggatt.wtdepannmin,

muaggatt.wtdepaprjunmin,

muaggatt.flodfreqdcd,

muaggatt.flodfreqmax,

muaggatt.pondfreqprs,

muaggatt.aws025wta,

muaggatt.aws050wta,

muaggatt.aws0100wta,

muaggatt.aws0150wta,

muaggatt.drclassdcd,

muaggatt.drclasswettest,

muaggatt.hydgrpdcd,

muaggatt.iccdcd,

muaggatt.iccdcdpct,

muaggatt.niccdcd,

muaggatt.niccdcdpct,

muaggatt.engdwobdcd,

muaggatt.engdwbdcd,

muaggatt.engdwbll,

muaggatt.engdwbml,

muaggatt.engstafdcd,

muaggatt.engstafll,

muaggatt.engstafml,

muaggatt.engsldcd,

muaggatt.engsldcp,

muaggatt.englrsdcd,

muaggatt.engcmssdcd,

muaggatt.engcmssmp,

muaggatt.urbrecptdcd,

muaggatt.urbrecptwta,

muaggatt.forpehrtdcp,

muaggatt.hydclprs,

muaggatt.awmmfpwwta

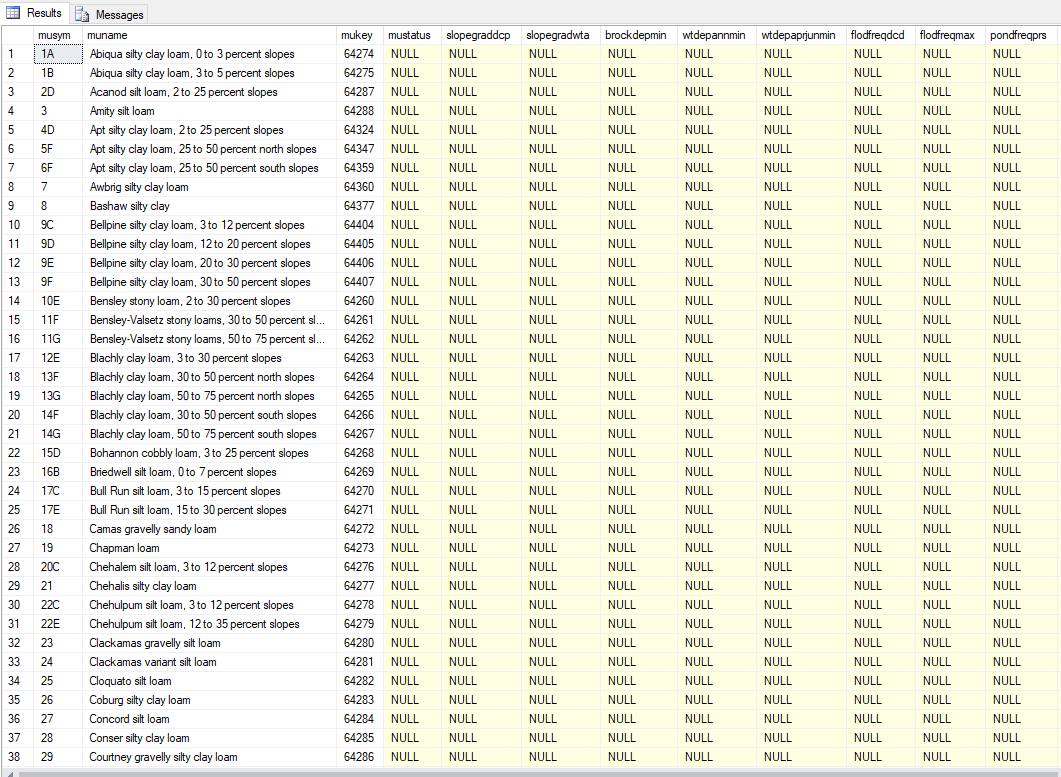
FROM

mapunit temp\_mapunit

INNER JOIN mapunit ON temp\_mapunit.mukey = mapunit.mukey

LEFT JOIN muaggatt ON muaggatt.mukey is null

The output for this query is 939 rows. As the code tells us, this creates a skeleton for the DataTable called “muaggatt”. This DataTable will be used as input to other methods. Some of the output is shown below. SSMS would be needed to see the entire dataset. Notice that the only columns populated will be musym, muname and mukey. The rest of the columns have NULL as values. The first 38 rows are shown below; not all columns are shown due to space limitations in this document.



Note that I will refer to the output of 939 rows as the “muaggatt DataTable”. This is passed into methods using the text “muaggatt”. An example is shown below where “muaggatt” is passed into the method aggAtts.DeriveAggAtts() as “aggAtts.DeriveAggAtts(**muaggatt**)”.

**Method LoadMuaggatt() calls method DeriveAggAtts()**.

private void **LoadMuaggatt**()

//lines 2645-2654

DataTable muaggatt = \_db.ExecuteDataTableQuery(commandText, "Creating Muaggatt data");

MapunitAggregatedAttributes aggAtts = new MapunitAggregatedAttributes(\_db);

//Calculate the column values and load the muaggatt table

**aggAtts.DeriveAggAtts(muaggatt); //LoadMuaggatt() calls DeriveAggAtts()**

IList<SqlParameter> parms = new List<SqlParameter>();

SqlParameter p = new SqlParameter("@DataTable", muaggatt);

p.TypeName = "MuAggAttTable";

parms.Add(p);  
**//The Stored Procedure is called here:**

\_db.ExecuteProcedure("**SDMDM\_InsertMuaggatt**", parms, "Loading Muaggatt table");

So from the above code, the LoadMuaggatt() method calls method DeriveAggAtts() and when DeriveAggAtts() is done, **LoadMuaggatt() calls the “SDMDM\_InsertMuaggatt” Stored Procedure**.

The muaggatt column values are populated under the DeriveAggAtts() method in file

“C:\SoilDataManagerService\_Folder\trunk\WorkItems\MapunitAggregatedAttributes.cs”

public void **DeriveAggAtts**(DataTable muaggatt)

//line39

SDVEngineUtilities.SDVAggregationParameters aggParms = **GetAggParameters**(specs);

//line 53

DataSet aggs = **SDVEngineUtilities.RunAggregation**(\_sdmDB.Connection, SDVEngineUtilities.SDVDatabaseModel.SoilDataMart, \_sdvAttributes, aggParms, mukeys);

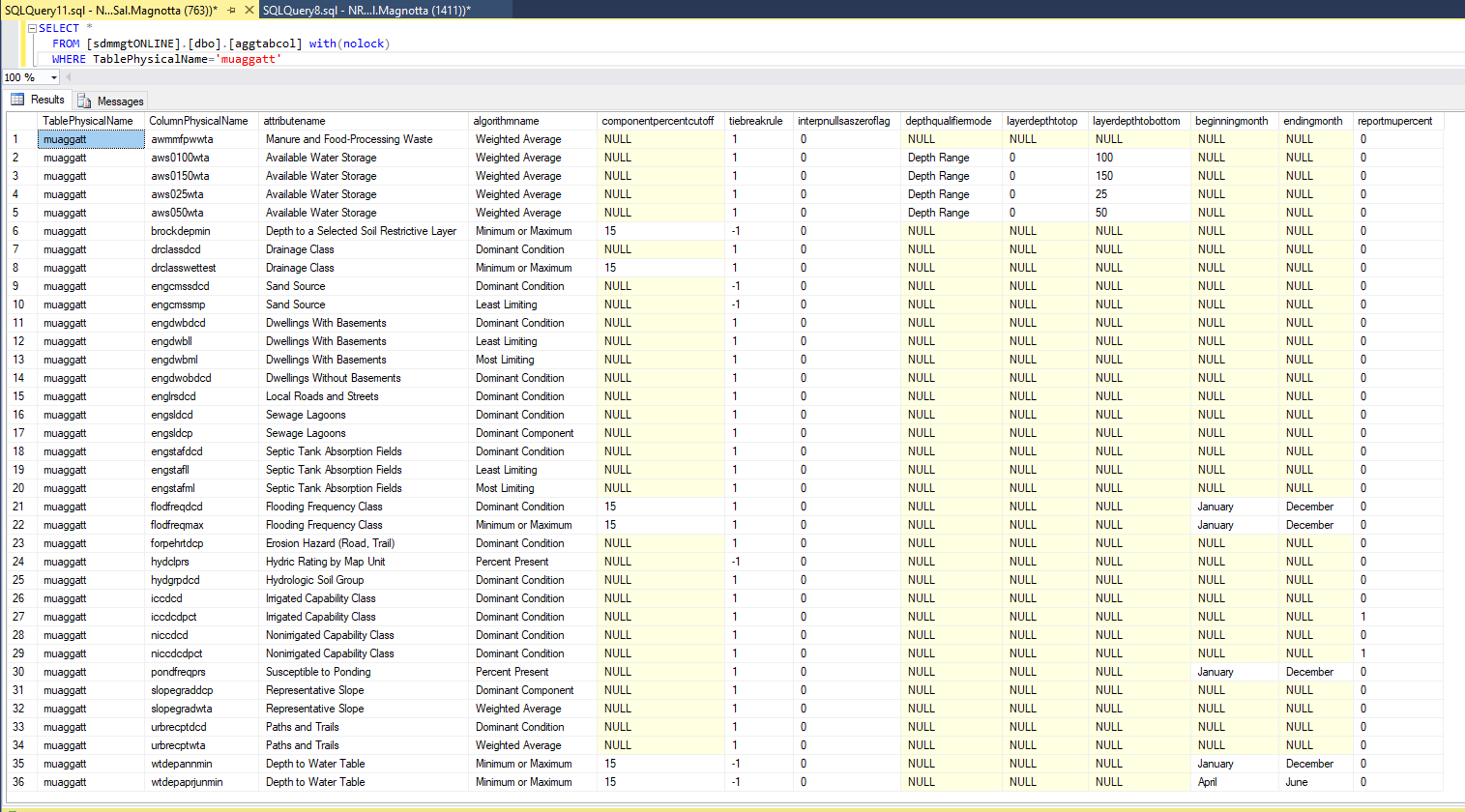
So the call structure (stack trace) is:   
Run() -> TransferArea()-> LoadMuaggatt() -> DeriveAggAtts()

DeriveAggAtts() calls GetAggParameters() and then SDVEngineUtilities.RunAggregation()

After DeriveAggAtts() finishes its work, it returns to the LoadMuaggatt() method and the stored procedure "**SDMDM\_InsertMuaggatt**" is called.

The following is a description of the **DeriveAggAtts**() method which will drive most of the work. I will indicate the end of this method in the discussion below.

Method DeriveAggAtts() first gets the set of aggregation specs for the muaggatt table by connecting to the SDMMgt database (which is a snapshot of the SDMMgtOnline database) and running a query against the “aggtabcol” table. This was one of the tables I mentioned at the start of this document.

The query is: select \* from aggtabcol where TablePhysicalName='muaggatt'  
What is known as the muaggatt table’s “specs” (specifications) is shown below. This output will be referenced in this document.  


This query returns 36 rows. One of the important columns is “attributename”. I believe this is also known as a “Rule”. Note that the code loops through all these 36 rows, one row at a time, using a while loop. I am only going to explain what happens with the first row of data in the above output.

Next, we need to query the same database (SDMMgt) and get the “sdvattribute” table details based on the muaggatt table’s specs’ “attributename” column.

The query in plain English is:

select \* from sdvattribute where attributename=each output row’s “attributename” column

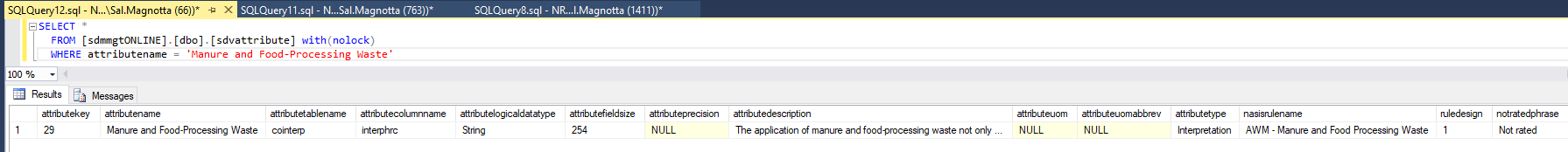
In other words, the first row of output above has an attributename of “Manure and Food-Processing Waste” so the query is SQL is:

SELECT \*

FROM [sdmmgtONLINE].[dbo].[sdvattribute] with(nolock)

WHERE attributename = 'Manure and Food-Processing Waste'

One row of output is returned as shown in the screen shot below. Note that the table sdvattribute has a lot of columns, so it is not practical to show all the details in this document. You need to run SSMS to see the details. Below is a screen shot of the partial data output when this query is run. Scrolling to the right will reveal the other column values. Notice for the output below the “attributetype” is “Interpretation” (not seen in the screen shot below). There are only two possible values for “attributetype”. They are “Interpretation” and “Property”. In the code, this output is known as the DataRow variable “\_sdvAttributes”.



I created another screen shot of the same output here, so the first column values can be seen:



Next, the code in method **DeriveAggAtts**() wants to save some of the sdvAttribute table’s column information for this one row of output above. The columns needed are attributecolumnname, attributetablename and attributelogicaldatatype. These values are shown below.

  
Next, the following method is called: **GetAggParameters**() with a parameter of “specs”. Remember that “specs” is the muaggatt table’s “specifications”.

The GetAggParameters() method retrieves a variety of parameter values to input into the SDVEngineUtilities.RunAggregation() method. I will place our example’s values for the code below.

//Code

aggParms.AggregationMethod = specs["algorithmname"]

aggParms.ComponentPercentCutoff = specs["componentpercentcutoff"]

aggParms.TiebreakRule = specs["tiebreakrule"]

aggParms.InterpretNullsAsZero = specs["interpnullsaszeroflag"]

aggParms.Layers = specs["depthqualifiermode"]

aggParms.DepthTop = specs["layerdepthtotop"]

aggParms.DepthBot = specs["layerdepthtobottom"]

aggParms.MonthBeg = specs["beginningmonth"]

aggParms.MonthEnd = specs["endingmonth"]

aggParms.DepthUOM = \_sdvAttributes["layerdepthuom"]

//Our example’s values (look at the first row in the muaggatt table’s “specs” output of 36 rows above)

aggParms.AggregationMethod = “Weighted Average”

aggParms.ComponentPercentCutoff = NULL

aggParms.TiebreakRule = 1

aggParms.InterpretNullsAsZero = 0

aggParms.Layers = NULL

aggParms.DepthTop = NULL

aggParms.DepthBot = NULL

aggParms.MonthBeg = NULL

aggParms.MonthEnd = NULL



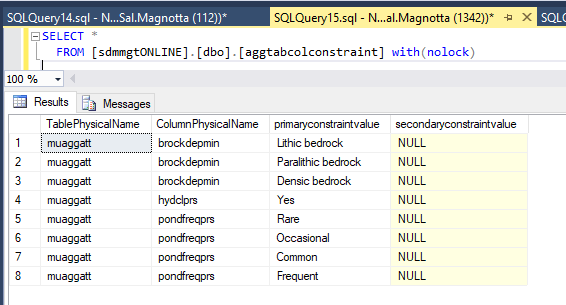
aggParms.DepthUOM = NULL



Next, GetAggParameters() retrieves constraint values from the child table of “aggtabcol”. This table is called “aggtabcolconstraint”. Notice that this table contains only 8 rows in total as shown below.

SELECT \*

FROM [sdmmgtONLINE].[dbo].[aggtabcolconstraint] with(nolock)



The code executes a query in which it retrieves the TablePhysicalName and ColumnPhysicalName from our “specs” output. The following is a rough SQL SELECT statement:

select \* from aggtabcolconstraint where TablePhysicalName = specs["TablePhysicalName"] AND

ColumnPhysicalName = specs["ColumnPhysicalName"]

We know the values of these two columns from our muaggatt table’s “specs” as shown below.



Therefore, we want to run this SQL SELECT statement in SSMS:

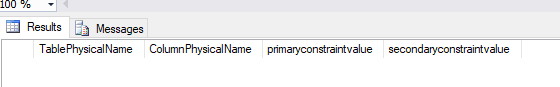
SELECT \*

FROM [sdmmgtONLINE].[dbo].[aggtabcolconstraint] with(nolock)

WHERE TablePhysicalName = 'muaggatt' AND

ColumnPhysicalName = 'awmmfpwwta'

…and that returns no rows of data as shown in the screen shot below (you can look in the 8 rows of output above and can see there is no data for these two values).



Therefore, the PrimaryConstraint and SecondaryConstraint values are zero-length strings. In code, this is represented as:

aggParms.PrimaryConstraint = new string[0];

aggParms.SecondaryConstraint = new string[0];

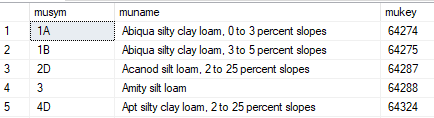
We go back from this method to the **DeriveAggAtts**() method. We still need to call the SDVEngineUtilities.RunAggregation() method which is in a for loop. But before that, let’s deal with a flag we need to set to true or false based on the muaggatt spec’s output above. We need to find out the value for reportmupercent. Looking at the first row of data in the 36 rows of output, this value is 0.



The flag’s name in the code is called “reportMuPct”. The code logic states that if the “reportmupercent” value is 0 or NULL then we set the flag “reportMuPct” to false. If “reportmupercent” is set to the value of 1, we set “reportMuPct” to true. In our case, as shown above, reportmupercent’s value is zero, so “reportMuPct” is set to false. We will need this later for some processing.

Next in the code is a “for loop”, used to go through all the rows in our “muaggatt DataTable”. Remember that table had 939 rows. Because of the limitations of the SQL “IN” clause, the code goes through the rows with a maximum of 500 rows at a time. This means the “mukeys” array will hold 500 mukey values, “do some processing”, and then when it goes the top of the for loop again, the code will process the rest of the rows which is 939-500 = 439 rows. So we process 500 rows then process 439 rows. The “do some processing” part is going to be a call to the method SDVEngineUtilities.RunAggregation() and then from that method’s Results DataTable output, we will be populating the muaggatt table (our “muaggatt DataTable” referenced above).

An example of what the first 5 values for the first 500 rows will be shown below. First, let’s capture a screen shot of those rows here.



So the first 5 populated values for the “mukeys” array are shown below.

mukeys[0] = 64274

mukeys[1] = 64275

mukeys[2] = 64287

mukeys[3] = 64288

mukeys[4] = 64324

This mukeys array will be passed as a parameter (as input) to the method SDVEngineUtilities.RunAggregation().

In fact, that method is called next in the code as shown below:

DataSet aggs = SDVEngineUtilities.RunAggregation(\_sdmDB.Connection, SDVEngineUtilities.SDVDatabaseModel.SoilDataMart, \_sdvAttributes, aggParms, mukeys);

The parameters for the method SDVEngineUtilities.RunAggregation() have already been populated by previous code and explained in this document, namely \_sdvAttributes, aggParms, and mukeys. The first two parameters specify that we are using the SDMOnline database and the SoilDataMart data model.

Where is the method SDVEngineUtilities.RunAggregation()? In the SoilDataManagerService solution, it is located at C:\SoilDataManagerService\_Folder\trunk\Dependencies\SDV\

GOV.USDA.NRCS.SoilDataViewer.SDVEngine.dll. The SDVEngine DLL is being used. This means to see the actual code we need to open the SDVEngine solution located at C:\SVN\SDV\SoilDataViewer\ SDVEngine.sln.

The SDVEngineUtilities.RunAggregation constructs queries on the fly based upon input parameters and mapunit data.  
  
In file “C:\SVN\SDV\SoilDataViewer\SDVEngine\clsSDVEngine.cs” lines 733 through 752 contain the RunAggregation() method. The code “!=” is read as “not equals”.

// This is the one that actually does the work.

private static DataSet RunAggregation(DbConnection db, SDVDatabaseModel DBModel, DataRow DRRule, SDVAggregationParameters AP, int[] mukeys, string distmdID)

{

DataSet ds = null;

try

{

ds = FetchData(db, DBModel, DRRule, AP, mukeys, distmdID);

ds = InitiallyAggregateData(ds, DRRule, AP);

ds = ReduceMultiples(ds, DRRule, AP);

if (AP.InterpretNullsAsZero == "true") ds = ConvertNullsToZeros(ds);

ds = AggregateData(ds, DRRule, DBModel, AP);

ds = FinishForDisplay(ds, db, DBModel, DRRule, AP, mukeys, distmdID);

}

catch (Exception ex)

{

if (ds != null) { ds.Dispose(); ds = null; }

throw new Exception("ERROR: Unable to perform aggregation.", ex);

}

return ds;

}

After RunAggregation() is done with its work, it returns a DataSet to the method DeriveAggAtt(), and the code in the DeriveAggAtts() method runs. This code is shown below. The “Results” DataTable retrieved from RunAggregation() is returned and used for the rest of the processing.

DataTable results = aggs.Tables["Results"];

string colName = specs["ColumnPhysicalName"].ToString();

foreach (DataRow row in results.Rows)

{

DataRow[] muaggattRow = muaggatt.Select("mukey='" + row["mukey"] + "'");

if (muaggattRow.Length > 0)

{

muaggattRow[0][colName] = reportMuPct ? row["pctofmapunit"] : row["rating"];

}

}

<The method **DeriveAggAtts**() ends here>

At this point, the calculation of data into the “muaggatt” table’s columns is complete. The method DeriveAggAtts() ends and code execution continues in the LoadMuaggatt() method.  
  
The LoadMuaggatt() method calls the Stored Procedure “SDMDM\_InsertMuaggatt” to load the muaggatt table. The code is shown below.

IList<SqlParameter> parms = new List<SqlParameter>();

SqlParameter p = new SqlParameter("@DataTable", muaggatt);

p.TypeName = "MuAggAttTable";

parms.Add(p);

\_db.ExecuteProcedure("SDMDM\_InsertMuaggatt", parms, "Loading Muaggatt table");  
  
<Main Processing is finished>

**SDVEngine**

As far as the databases that can connect and use SDVEngine, they are NASIS (nasis), SoilDataMart (sdm),

Staging (sdwstaging) and MSAccess (e.g. soildb.mdb).  
  
There is a test project in the SDVEngine solution called SDVEngineTest. It uses an input XML file to run the SDVEngine. The file name is “TestCases.xml”. Here is the format of a record in the XML file.

<TestCases>

<Case Name="xxx"> (case name listed in console output)

<SDVRule Name="" /> (name of rule in sdvattribute table)

<Database Model="" /> ("NASIS", "SoilDataMart", "Staging" or "MSAccess")

<Mukey Value="" /> (mapunit to run aggregation on)

<Rating Value="" /> (expected value of rating)

<Percent Value="" /> (expected value for percent of mapunit)

<PrimaryConstraint> (optional)

<Value>zzz</Value> (one or more value for primary constraint column)

</PrimaryConstraint>

<SecondaryConstraint Value="" /> (optional)

<AggregationMethod Value="" /> (name of aggregation method)

<ComponentPercentCutoff Value="" /> (optional: 0 to 100)

<TiebreakRule Value="" /> (optional: "high" or "low")

<InterpretNullsAsZero Value="" /> (optional: "true" or "false")

<LayersSelection Value="" /> (optional: "surface layer", "depth range" or "all layers")

<DepthRange Top="" Bottom="" /> (optional: depths in cm)

<MonthRange Begin="" End="" /> (optional: use "january" etc.)

</Case>

<Case ...

</TestCases>

These 2 inputs are Rating Value and Percent value. An example is:

<Rating Value="Well drained" />

<Percent Value="96" />

For the rating value, the comment states "expected value of rating" and for the percent value, the comment states "expected value for percent of mapunit". I decided to run through the code with my example input and have blank information for the "rating" and "percent" values. After code execution, the expected values were printed out. They were "0.335" for Rating Value and "90" for Percent Value.

<Case Name="ProcessingWaste">

<SDVRule Name="Manure and Food-Processing Waste" />

<Database Model="SoilDataMart" />

<Mukey Value="64274" />

<AggregationMethod Value="weighted average" />

<TiebreakRule Value="high" />

<InterpNullsAsZero Value="false" />

<Rating Value="" />

<Percent Value="" />

</Case>

I ran SDVEngineTest via "Run Selected Test" and received "Completed test with rating = 0.335 and percent = 90". Now I have the two inputs I was looking for. In this case, I am running against mapunitkey = 64274, mapunitsymbol="1A" and mapunitname="Abiqua silty clay loam, 0 to 3 percent slopes".

There is a "driver" method called RunAggregation() that is called.

Tables in returned DataSet:

- Results (areasymbol areaname saversion saverest mukey musym nationalmusym muname rating pctofmapunit)

The following tables are also in the returned DataSet for non-class soil interpretations only:

- Components (mukey cokey compname comppct\_r interphr) with relationship ResultComponents to Results.

- Reasons (cokey interphrc interphr) with relationship ComponentReasons to Components.

The first method within RunAggregation() to be called is FetchData(). This was noted a few pages back. Regarding FetchData(), “This private method fetches all of the data to be aggregated, constructing the appropriate query first”.

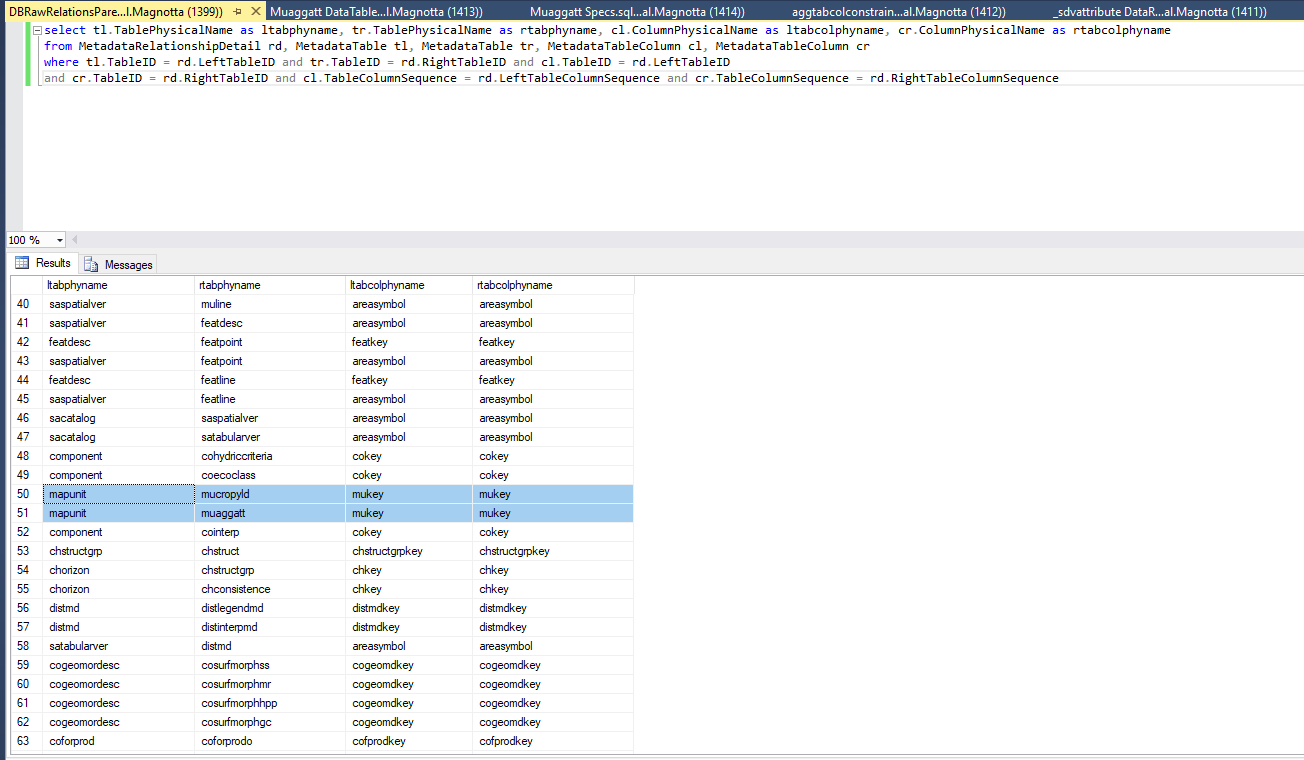
XmlDocument XMLDBRelations = GetDBRelationsSQL(db, DBModel);  
GetDBRelationsSQL() methods' description:

These public methods create an XML document containing the SQL From and Where clauses needed to relate the mapunit table and its descendents contained within the database specified by the given database connection or database connection string.

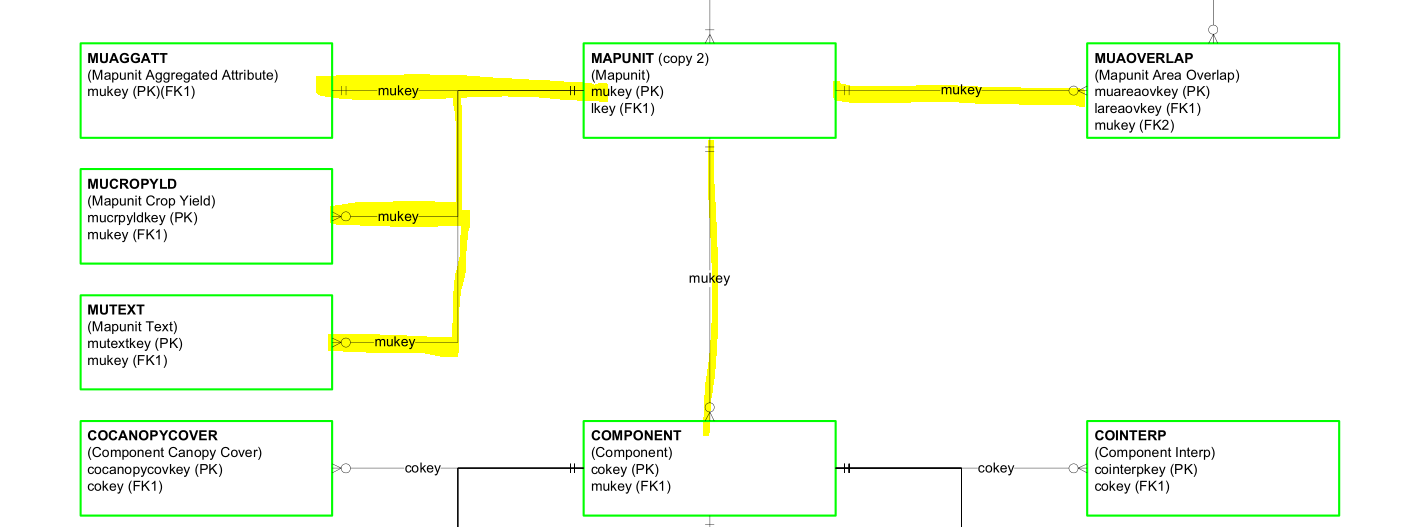
In method GetDBRelationsSQL(), the first query run is:

select tl.TablePhysicalName as ltabphyname, tr.TablePhysicalName as rtabphyname, cl.ColumnPhysicalName as ltabcolphyname, cr.ColumnPhysicalName as rtabcolphyname from MetadataRelationshipDetail rd, MetadataTable tl, MetadataTable tr, MetadataTableColumn cl, MetadataTableColumn cr where tl.TableID = rd.LeftTableID and tr.TableID = rd.RightTableID and cl.TableID = rd.LeftTableID and cr.TableID = rd.RightTableID and cl.TableColumnSequence = rd.LeftTableColumnSequence and cr.TableColumnSequence = rd.RightTableColumnSequence

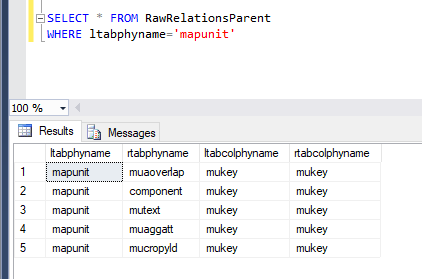
…which produces the following:



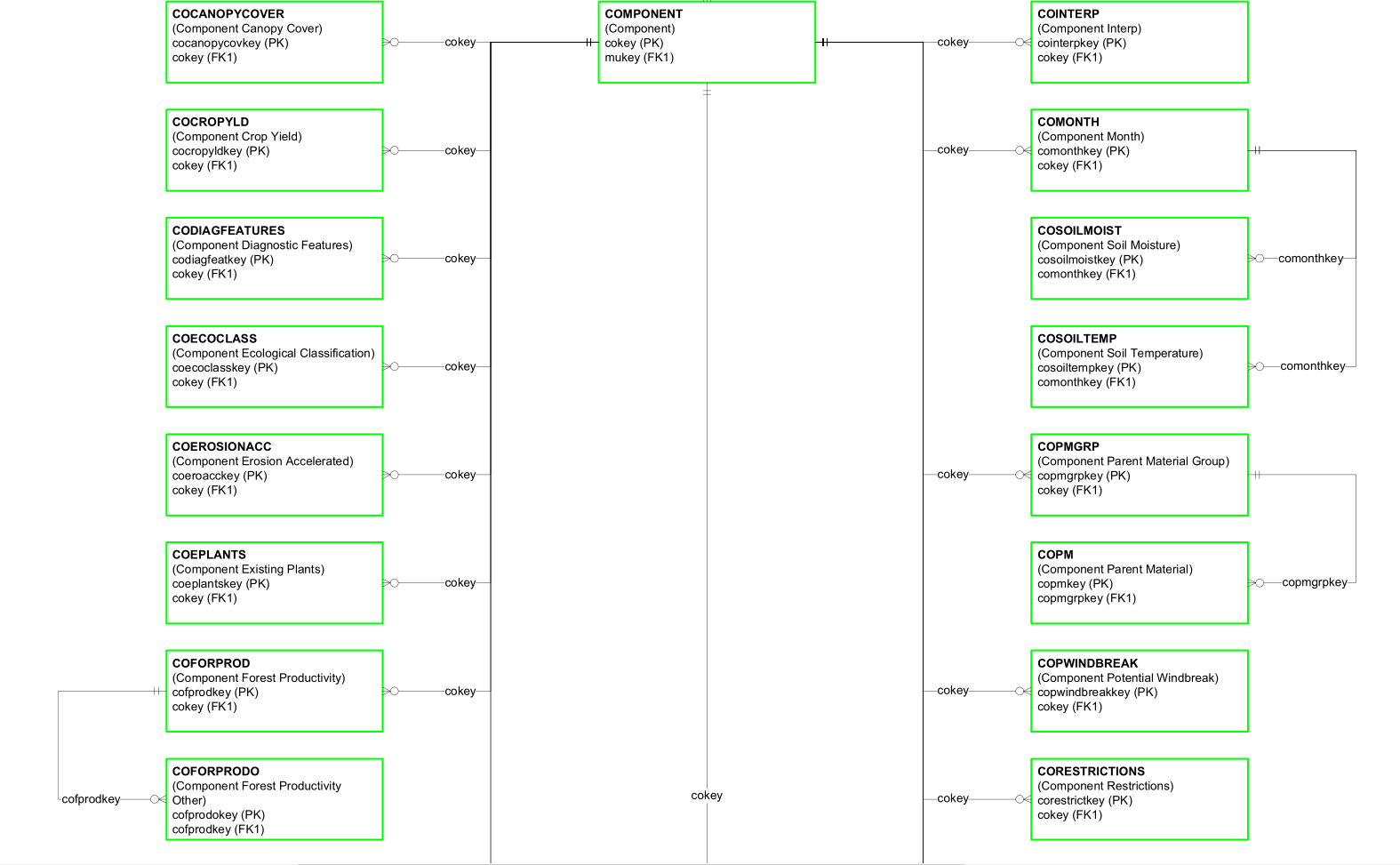
The SQL is finding the relationships between tables. For example, the diagram below displays the mapunit table’s children (5 tables).

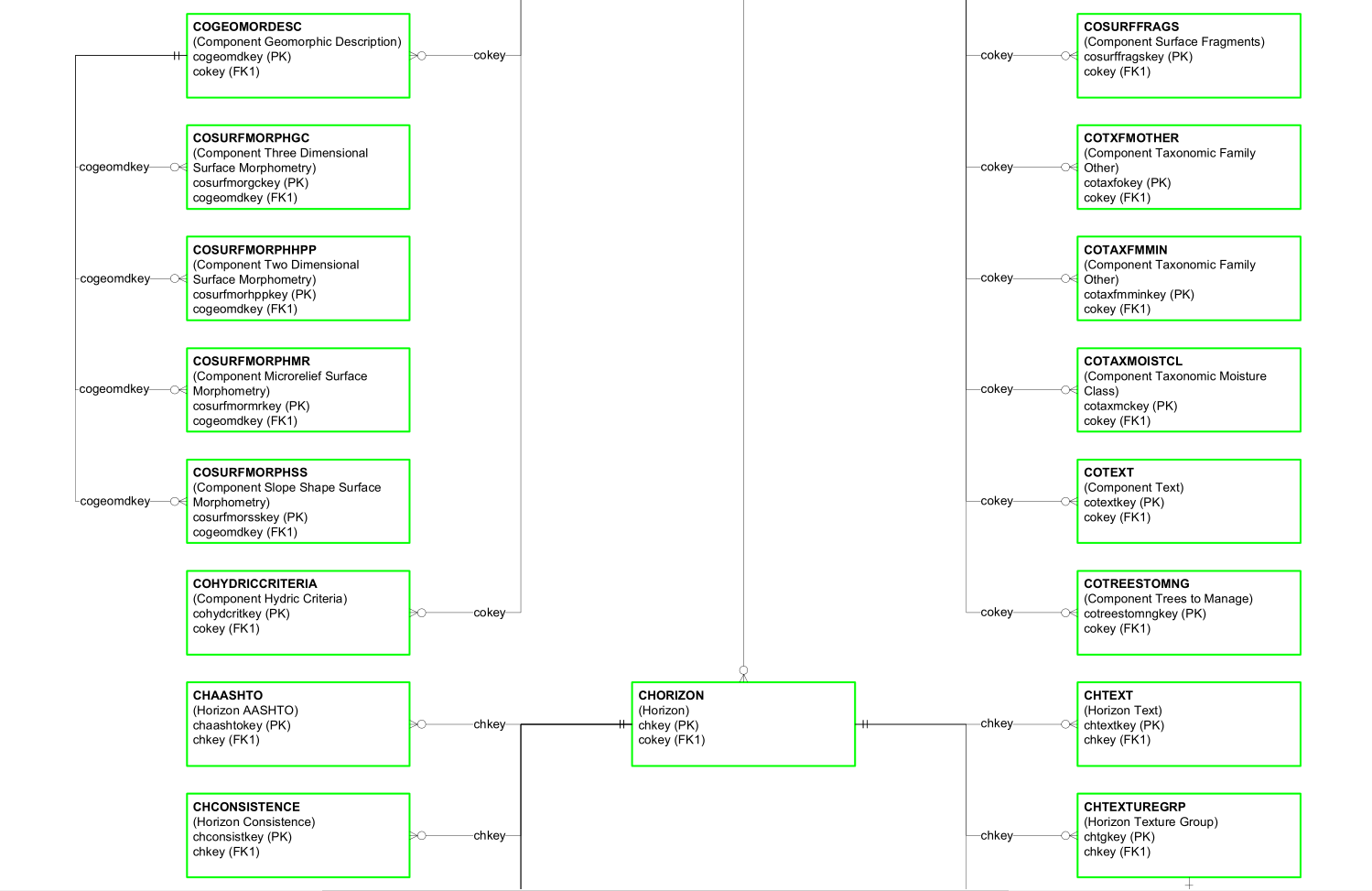


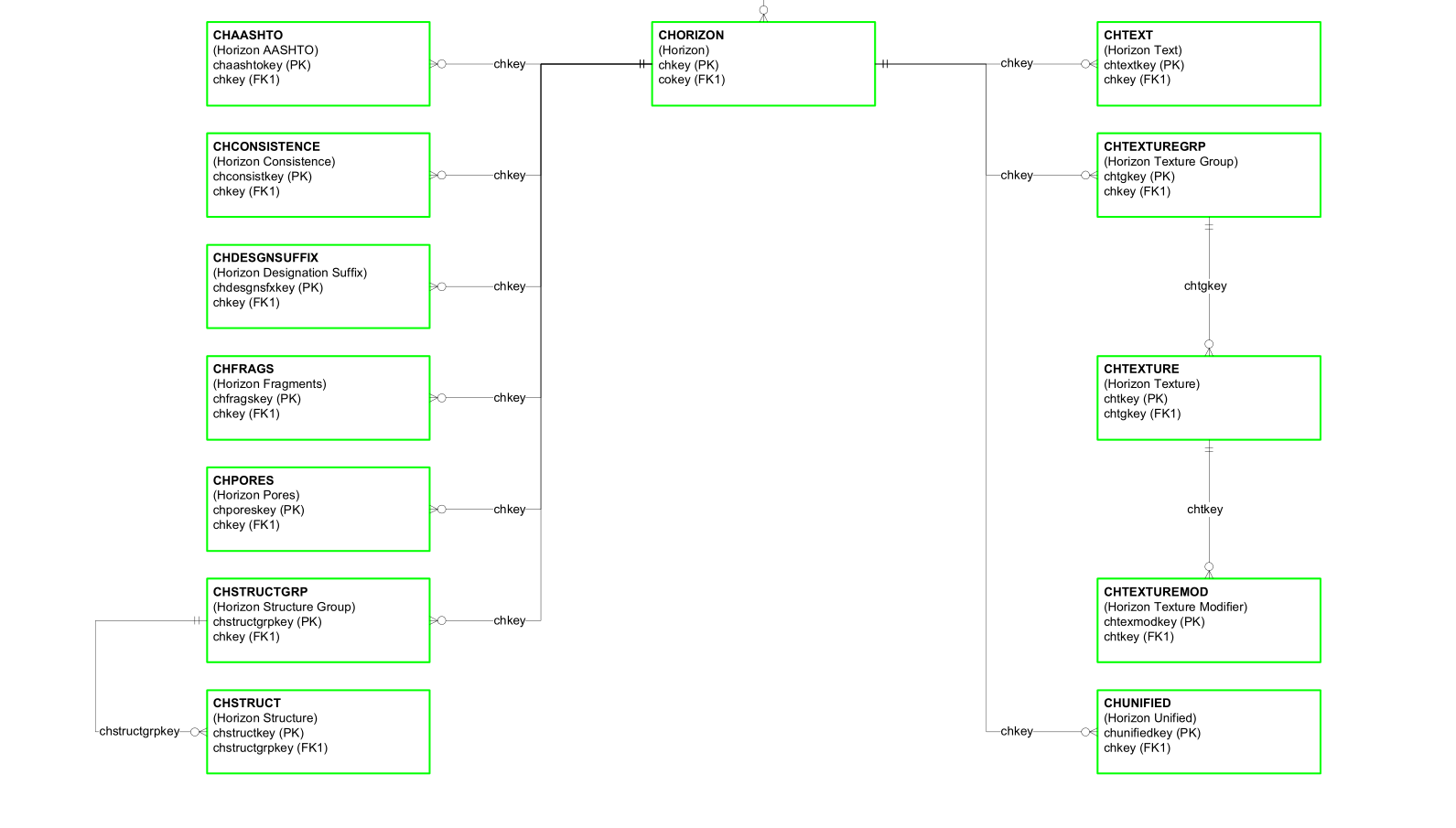
Next, the method GetDBChildRelationsSQL() is recursively called to get the relationships of all child tables while creating an XML file.  
  
The GetDBChildRelationsSQL() private method recursively populates the given XML document with the SQL from and where clauses needed to relate the tables specified within the given data table of raw database table relationships.  
  
The following is a sample of one of the queries and results the GetDBChildRelationsSQL() produced. As you can see, there are 5 child tables for the mapunit “parent” table. The “RawRelationsParent” table below is a temporary table.



The 5 child tables for parent table “mapunit” are “muaoverlap”, “component”, “mutext”, “muaggatt” and “mucropyld”.

The code then does the same thing for each of the 5 child tables above recursively. This means the code finds all descendants of the parent “mapunit” table. In other words, find all descendants for “muaoverlap”, “component”, …, and so on, up to “mucropyld”. Depending upon the database structure, this can yield many tables since we are requesting all descendant tables.  
  
Notice in the data model that all 5 tables do not have children, except for the “component” table. There are many descendant tables for “component”, so please refer to the data model.   
  


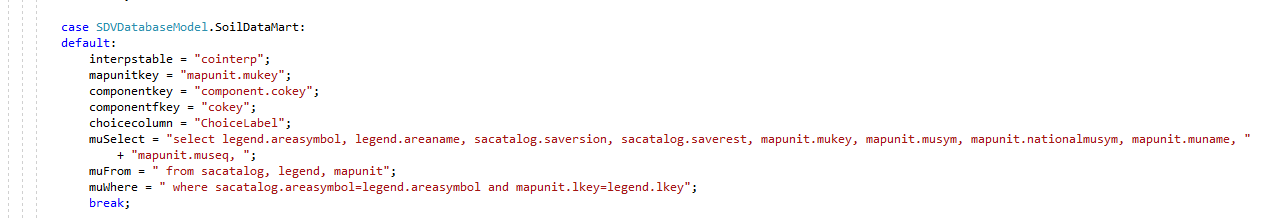




I am running against mapunitkey = 64274, mapunitsymbol="1A" and mapunitname="Abiqua silty clay loam, 0 to 3 percent slopes".

Next, two queries are going to be produced through the code execution in FetchData(), “query” and “unionquery”. Both queries are dynamically created.

We are calling SDVEngine from the sdm database, so we start with the following variables.



The “query” is produced from select clause additions for attribute, constraints and tiebreakers. Then the where clause is built from additional conditions. This is all concatenated together to produce the “query”:

select legend.areasymbol, legend.areaname, sacatalog.saversion, sacatalog.saverest, mapunit.mukey, mapunit.musym, mapunit.nationalmusym, mapunit.muname, mapunit.museq, cointerp.interphr as attributevalue, cointerp.interphr as tiebreakvalue, null as primaryconstraint, null as secondaryconstraint, component.cokey as cokey, component.compname, component.comppct\_r, cointerp.interphr **into #sdvmain** from sacatalog, legend, mapunit, component, cointerp where sacatalog.areasymbol=legend.areasymbol and mapunit.lkey=legend.lkey and component.mukey=mapunit.mukey and cointerp.cokey=component.cokey and mapunit.mukey in ('64274') and cointerp.mrulename = 'AWM - Manure and Food Processing Waste' and cointerp.ruledepth = 0 and component.comppct\_r is not null and component.comppct\_r > 0

Notice that a temporary table is being used called “sdvmain” above (in the SQL above, look for “#sdvmain”).

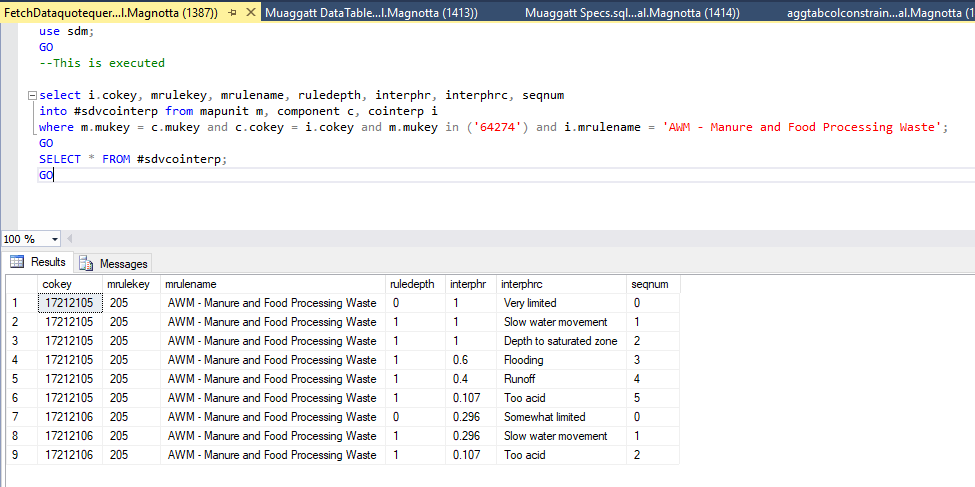
We are not done constructing the SQL code.

The ”unionquery” produced through the dynamic code is:

union select legend.areasymbol, legend.areaname, sacatalog.saversion, sacatalog.saverest, mapunit.mukey, mapunit.musym, mapunit.nationalmusym, mapunit.muname, mapunit.museq, null as attributevalue, null as tiebreakvalue, null as primaryconstraint, null as secondaryconstraint, null as cokey, null as compname, null as comppct\_r, null as interphr from sacatalog, legend, mapunit where sacatalog.areasymbol=legend.areasymbol and mapunit.lkey=legend.lkey and mapunit.mukey in ('64274') and mapunit.mukey not in (select mukey **from #sdvmain**) order by areasymbol, museq

Notice the “unionquery” also uses the “sdvmain” temporary table.

Fetch the data to be aggregated and set up the DataSet to be returned. If the Soil Data Mart SQL Server database is being used (in this case it is), select any needed records out of the cointerp table into a temp table first (and change the main query to use that temp table) because of the size of the cointerp table in that national database. The cointerp table is very large.  
  
Since we are using the sdm database, we must use temp tables. There is another string variable “quote” that will be dynamically created. The “quote” query is:

select i.cokey, mrulekey, mrulename, ruledepth, interphr, interphrc, seqnum into **#sdvcointerp** from mapunit m, component c, cointerp i where m.mukey = c.mukey and c.cokey = i.cokey and m.mukey in ('64274') and i.mrulename = 'AWM - Manure and Food Processing Waste'  
  
This “quote” query places data into a temporary table called “sdvcointerp”. As seen below, this returns 9 rows.  
  


We need to replace the “cointerp” table in the “query” above with the temporary table “sdvcointerp”. We do this through the following code:

query = query.Replace("cointerp", "#sdvcointerp");

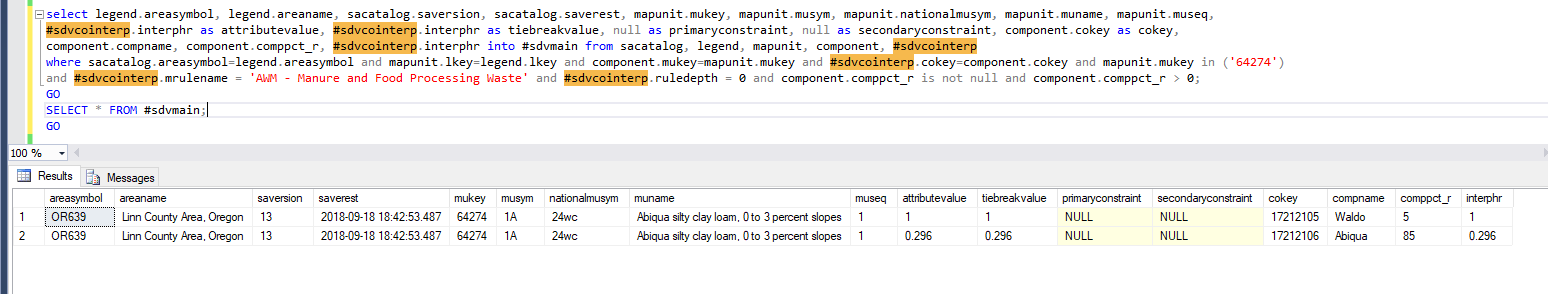
OK, so what does “query” look like now? See directly below. Compare what “query” was above to what is shown below. Notice how “cointerp” has been replaced with “sdvcointerp”. Also notice the temporary table “sdvmain”.  
  
select legend.areasymbol, legend.areaname, sacatalog.saversion, sacatalog.saverest, mapunit.mukey, mapunit.musym, mapunit.nationalmusym, mapunit.muname, mapunit.museq, **#sdvcointerp**.interphr as attributevalue, **#sdvcointerp**.interphr as tiebreakvalue, null as primaryconstraint, null as secondaryconstraint, component.cokey as cokey, component.compname, component.comppct\_r, **#sdvcointerp**.interphr **into #sdvmain** from sacatalog, legend, mapunit, component**, #sdvcointerp** where sacatalog.areasymbol=legend.areasymbol and mapunit.lkey=legend.lkey and component.mukey=mapunit.mukey and **#sdvcointerp.**cokey=component.cokey and mapunit.mukey in ('64274') and **#sdvcointerp**.mrulename = 'AWM - Manure and Food Processing Waste' and **#sdvcointerp**.ruledepth = 0 and component.comppct\_r is not null and component.comppct\_r > 0

The query directly above produces 2 rows of data.

The “query” variable is set back to this short string in the code:  
query = "select \* from #sdvmain ";

Next, query plus unionquery is combined (concatentated) to produce another query to be run:

query + unionquery  
  
select \* from #sdvmain union select legend.areasymbol, legend.areaname, sacatalog.saversion, sacatalog.saverest, mapunit.mukey, mapunit.musym, mapunit.nationalmusym, mapunit.muname, mapunit.museq, null as attributevalue, null as tiebreakvalue, null as primaryconstraint, null as secondaryconstraint, null as cokey, null as compname, null as comppct\_r, null as interphr from sacatalog, legend, mapunit where sacatalog.areasymbol=legend.areasymbol and mapunit.lkey=legend.lkey and mapunit.mukey in ('64274') and mapunit.mukey not in (select mukey from #sdvmain) order by areasymbol, museq  
  
This results in the following 2 rows of data:

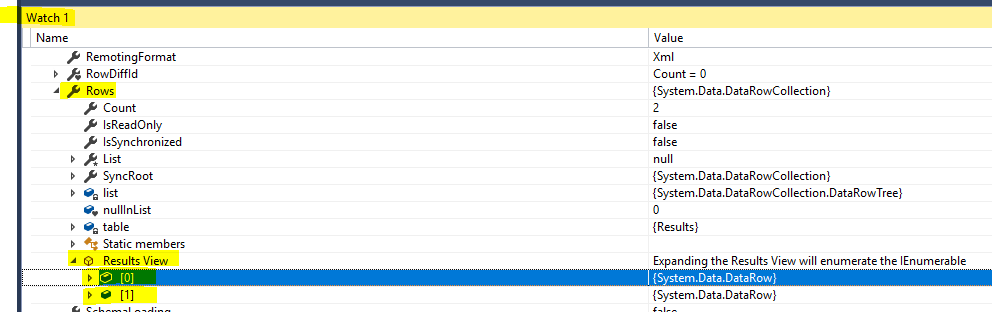
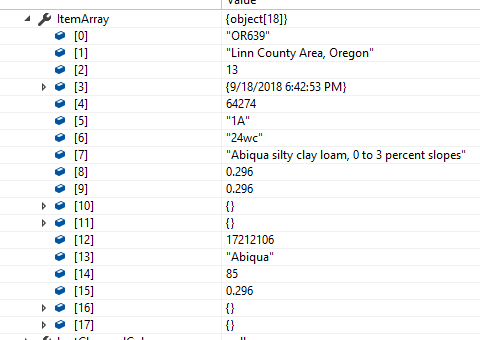


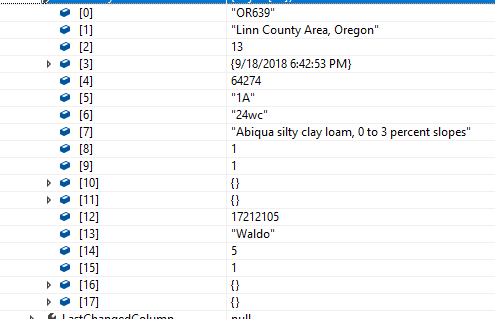
We are using temporary tables, so the next part of the code does some cleanup.

string cmd = "drop table #sdvmain; drop table #sdvcointerp";

ds.Tables["Results"] shows 2 rows of data below.

2 rows of data in the Results Data Table (same as the 2 rows of data above) are shown below. To get to this in Visual Studio, go to the “Watch1” tab and type in ds.Tables["Results"]. After that drill down to (expand) Rows >> Results View. You will see two rows labeled as [0] and [1]. Next, expand [0] >> ItemArray and [1] >> ItemArray. Each ItemArray will display the values for each of the rows as seen below.



The following is important in following the execution of the SDVEngine code. Note that two columns are added to the “Results” table. These are “rating” and “pctofmapunit”. This is the code that adds these two columns located at the end of the FetchData() method:

ds.Tables["Results"].Columns.Add("rating", Type.GetType("System.String"));

ds.Tables["Results"].Columns.Add("pctofmapunit", Type.GetType("System.Int32"));

The “rating” column is the “rating value” or expected value of rating. The “pctofmapunit” column is the “percent value” or expected value for percent of mapunit. Recall from above that “Results” is a data table returned by this SDVEngine. Notice the two columns listed at the end below.

- Results (areasymbol areaname saversion saverest mukey musym nationalmusym muname **rating pctofmapunit**)

The next method run is InitiallyAggregateData() called as seen below:

ds = InitiallyAggregateData(ds, DRRule, AP);

This private method performs any needed initial aggregation (up to the component level).

The comments in the code are:  
Find out if horizon aggregation is needed and, if so, do it.

For each row of data, if it hasn't already been marked for deletion, then find the effective depth range to aggregate within, compute a new aggregated attribute and tiebreak value for the row from all of the rows that correspond to the same component that the row corresponds to, and then mark each of those other rows used in that calculation for deletion. When finished, permanently delete all of the rows that were marked for deletion. Note that this ignores components that were added back (actualdepthtop is null) when looping but does not ignore them when it is time to mark rows for deletion.

In our case, we already know our “attributetable” is “cointerp” and not “chorizon”, so we are not going to trace through the horizon aggregation (horizon aggregation is not needed). So this means we are not doing anything in this method.

The code that does the check to determine if we should perform the horizon aggregation is as follows:

if (attributetable.ToLower() == "chorizon" && (attributetype == "integer" || attributetype == "float"))

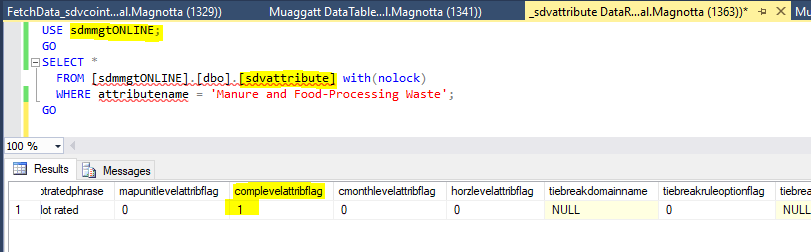
As you can see, the “attributetable” must be “chorizon” and the “attributetype” must either be integer or float. In our case “attributetable” is “cointerp” and the “attributetype” is “string”, so this check fails, and we do not do horizon aggregation.  
  
The next method to be called is ReduceMultiples(). This private method reduces multiple values in the data prior to aggregating. It reduces to a single row per mapunit or component depending on the level of the attribute being aggregated. It keeps the row with the highest/lowest non-null tiebreak value for each mapunit/component depending upon what the tiebreak rule is for the attribute being aggregated. If all tiebreak values for a mapunit/component are null, it keeps any one row for that mapunit/component.

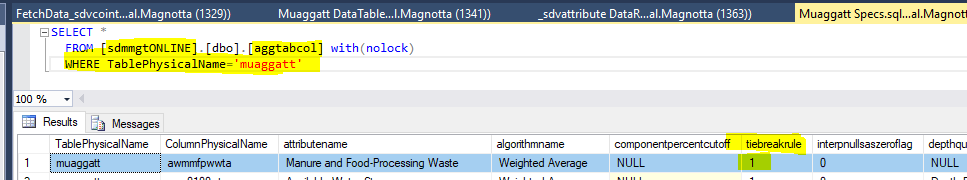
Using our 2 rows of data above from the “Results” table, the code performs a “SELECT” statement via the following code:

dt.SELECT("tiebreakvalue is not null", "mukey" + (cl ? ", cokey" : "") + ", tiebreakvalue " + (AP.TiebreakRule == "1" ? "desc" : "")

Notice we are checking for tiebreakvalue is NOT NULL.

The SELECT statement above is the DataTable.Select Method (dt stands for DataTable). The “cl” variable above evaluates whether the "complevelattribflag" is "1". In our case, it is 1 (see screen shot directly below), so the cl value is “true”.



Our “AP.TiebreakRule” is 1 as shown in the screen shot directly below.  


Select(String, String)

Gets an array of all DataRow objects that match the filter criteria, in the specified sort order.

public System.Data.DataRow[] Select (string filterExpression, string sort);

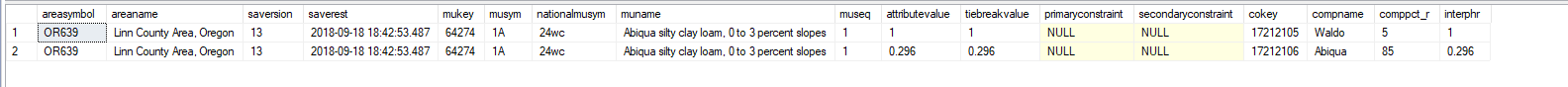
So the first parameter “tiebreakvalue is not null” is the “filterExpression” or the criteria to use to filter the rows. The second parameter translates to “mukey, cokey, tiebreakvalue desc” and is the string specifying the column and sort direction (descending in this case).

This translates to the more readable:   
  
SELECT \* FROM #sdvmain

WHERE tiebreakvalue is not null

ORDER BY mukey, cokey, tiebreakvalue desc

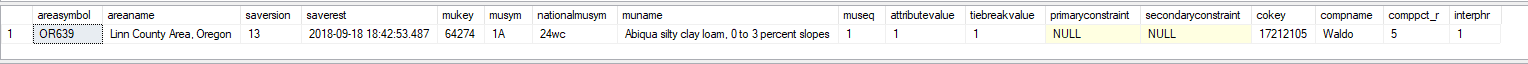
This returns the two rows we have seen before in our Results table. In this case, the SELECT statement did not do much because the “tiebreakvalue” is not null and the map unit key is the same for both rows (64274). For the two rows, the component keys (cokey) are different (17212105 and 17212106) and the tiebreakvalues are different (1 and 0.296).



Next, is a for loop to go through each of these rows and run the following code:

dt.Select("mukey='" + r["mukey"].ToString() + (cl ? "' and cokey='" + r["cokey"].ToString() + "'" : "'"));  
  
This translates to the more readable:   
For the first row:  
SELECT \* FROM #sdvmain

WHERE mukey=64274 and cokey=17212105

  
  
For the second row:  
SELECT \* FROM #sdvmain

WHERE mukey=64274 and cokey=17212106



For each of these rows, the code checks if we should “keep” the row. It does this by checking if the row has been marked for deletion. In our case, both rows are kept.  
  
An almost identical block of code follows this, however this time we are checking for NULL (if tiebreakvalue is null). We already know our two rows of data both have tiebreakvalues, so this block of code is not executed. If it were executed and we had one or more null values for tiebreakvalue, then for each null-valued row, the code checks if we should “keep” the row. It does this by checking if the row has been marked for deletion.

dt.Select("tiebreakvalue is null", "mukey" + (cl ? ", cokey" : ""));

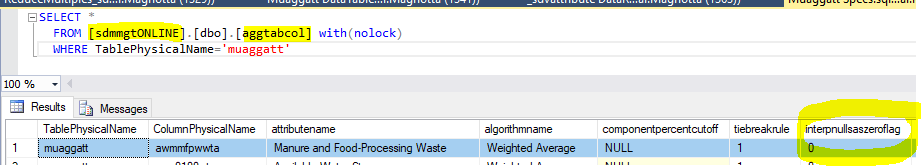
This translates to the more readable:   
  
SELECT \* FROM #sdvmain

WHERE tiebreakvalue is null

ORDER BY mukey, cokey

As discussed above, this SELECT statement returns no rows, because tiebreakvalue is not null.  
  


The next section of code calls the method ConvertNullsToZeros() if the column “interpnullsaszeroflag” is true or not. In our case, it is not true (it is 0 as shown in the screen shot below), so the method ConvertNullsToZeros() is not executed (it is skipped).

if (AP.InterpretNullsAsZero == "true") ds = ConvertNullsToZeros(ds);  
  


Next, we call the AggregateData() method as shown below.

ds = AggregateData(ds, DRRule, DBModel, AP);

The comment for this method is: This private method performs the final aggregation (to the mapunit level).  
  
In our case, the aggregation method is “weighted average”.

There are a lot of comments in this method. I will copy them here.

If no aggregation is to be performed (there is just one record per mapunit in this one case), then set the final rating value for each mapunit to its corresponding attribute value. The code for this is shown below. Notice that the two new columns added to the “Results” table (rating and pctofmapunit) are assigned. The “rating” column is set to the “attributevalue” column’s value and “pctofmapunit” is set to 100.

if (AP.AggregationMethod == "no aggregation necessary") {

foreach (DataRow r in DSData.Tables["Results"].Rows)

{

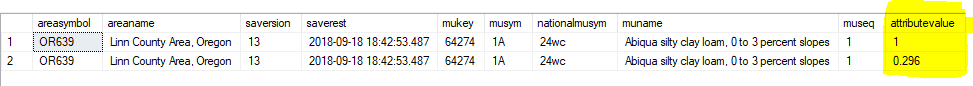
r["rating"] = r["attributevalue"];

r["pctofmapunit"] = 100;

}

}

Just as an example, recall what our data was. Our aggregation method in our case is not “no aggregation necessary”, so we skip the if clause above. I just wanted to show what “attributevalue” was referring to as shown in the screen shot directly below.



Tables in returned DataSet:

- Results (areasymbol areaname saversion saverest mukey musym nationalmusym muname rating pctofmapunit)

The following tables are also in the returned DataSet for non-class soil interpretations only:

- Components (mukey cokey compname comppct\_r interphr) with relationship ResultComponents to Results.

- Reasons (cokey interphrc interphr) with relationship ComponentReasons to Components.

If aggregation is to be performed (as in our case), start by modifying our existing dataset to separate the mapunits and components into two tables. A new table, Components, is added from a copy of the existing Results table. Any fake components (null cokey) created to add back a mapunit with no components are deleted. If percent composition is significant for the aggregation method, then all components with a null percent composition are also deleted. Then the existing Results table is modified to remove all but a single record per mapunit. Aggregation is then performed. Finally, the columns that are no longer needed in the Results table are deleted.

The code starts out by setting “dtM” (data table Mapunits) and “dtC” (data table Components) to the “Results” data table.

DataTable dtM = DSData.Tables["Results"];

DataTable dtC = DSData.Tables["Results"].Copy();

DataRow[] drs = null;

string aggc, mm, so;

Next, for each row in data table Mapunits, check that the row state is not “Deleted”. If the row state is not deleted, then set a data row array “drs” to SELECT \* FROM “datatable Mapunits” WHERE mukey=’64274’. Mark each of the rows returned from the SELECT statement for deletion via r.Delete(), but then reject all changes made via m.RejectChanges(). After the for loop ends, execute dtM.AcceptChanges() which commits the changes for the data table Mapunits. The result of running this code is that the Mapunits data table now only has one row in it, that is, the one with the cokey = 17212106. The code “!=” is read as “not equals”.   
  
foreach (DataRow m in dtM.Rows) {

if (m.RowState != DataRowState.Deleted) {

drs = dtM.Select("mukey='" + m["mukey"].ToString() + "'");

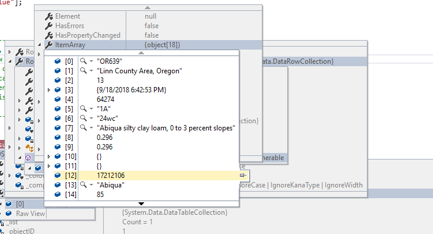
foreach (DataRow r in drs) r.Delete();

m.RejectChanges();

}//end if

}//end foreach

dtM.AcceptChanges();





Next, for “dtC” (data table Components), set its table name to “Comps”. Add the Comps (Components) data table to the main DataSet, DSData. This means the main DataSet, DSData, now consists of the “Results” and “Comps” data tables. Next, a “data relations” table called “MapUnitComps” is added to the main DataSet, DSData. This means the main DataSet, DSData, now consists of the “Results” and “Comps” data tables and the “MapunitComps” data relation. The parent table for this relation is the “Mapunit” data table and the child table for this relation is the “Comps” (Components) data table. The common column connecting these two data tables is “mukey” (mapunit key). The column “mukey” serves as the link between the two data tables. The last parameter is “false” which is a value that indicates whether to create constraints (in our case we are specifying not to create constraints). A DataRelation also has a Nested property which, when set to true in our case, causes the rows from the child table to be nested within the associated row from the parent table when written as XML elements using WriteXml .

dtC.TableName = "Comps";

DSData.Tables.Add(dtC);

DSData.Relations.Add("MapunitComps", dtM.Columns["mukey"], dtC.Columns["mukey"], false);

DSData.Relations["MapunitComps"].Nested = true;

I found that the “MapunitComps” data relation is only used if the aggregation method is “most limiting” and “least limiting” (these last two are processed together), “minimum or maximum”, “dominant component”, and “dominant condition”. Since our aggregation method is “weighted average”, this data relation will not be used.

Notice in the code directly below that the “FetchAllComps()” method is executed. This private method determines whether components with null percent composition should be used.

private static bool **FetchAllComps**(SDVAggregationParameters AP)

{

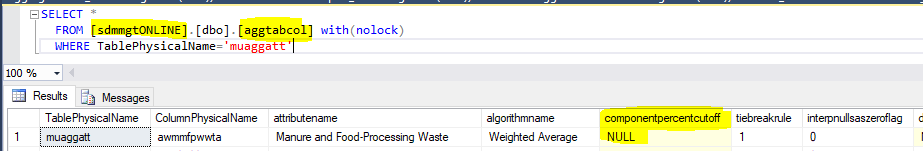
return (AP.AggregationMethod == "minimum or maximum" ||

AP.AggregationMethod == "most limiting" ||

AP.AggregationMethod == "least limiting") &&

AP.ComponentPercentCutoff.Length == 0;

}  
  
Our aggregation method is “weighted average” so the first three conditions above do not match so are all false. The last condition is if AP.ComponentPercentCutoff.Length == 0. We already know the componentpercentcutoff is NULL or zero length, so we have the Boolean condition (false AND true) which results in false. Therefore, in our case, the method FetchAllComps() returns false.

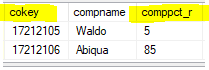


The SQL to run next is “SELECT \* FROM data table Comps WHERE cokey is null and comppct\_r is null”. Next, for each of the rows returned we mark them for deletion and accept (commit) the changes. We already know we have data for those two columns (cokey and comppct\_r) so no rows will be returned.

drs = dtC.Select("cokey is null" + (FetchAllComps(AP) ? "" : " and comppct\_r is null"));

foreach (DataRow r in drs) r.Delete();

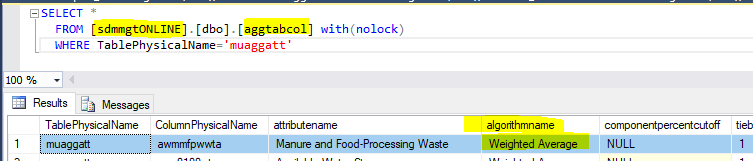
dtC.AcceptChanges();





The value for “AP.AggregationMethod” is compared and processed. In our case, the “AP.AggregationMethod” is “weighted average” as shown below.

aggParms.AggregationMethod = specs["algorithmname"];



The “algorithmname” or “Aggregation Method” can be one of several values. These are “no aggregation necessary”, “percent present”, “most limiting” and “least limiting” (these last two are processed together), “weighted average” (in our case, this is our aggregation method), “minimum or maximum”, “dominant component”, and “dominant condition”.   
  
Comments for aggregation method “percent present”:  
Percent present for the selected condition:

For each mapunit, the final rating value is the sum of the percent compositions of all components that remain after applying the Where conditions and constraints. The percent of mapunit column contains the same number.  
  
Comments for aggregation methods “most limiting” and “least limiting”:

Most/Least Limiting:

For each mapunit, set its final rating to the attribute value for its component with the highest or lowest tiebreak value. For the Most Limiting aggregation method, use the highest tiebreak if the attribute is a limitation (ruledesign=1) and use the lowest tiebreak if the attribute is a suitability (ruledesign=2). Do the opposite for the Least Limiting aggregation method. If some components are not rated (null attribute value) the result is also "not rated" unless there is a component that has the most extreme attribute value (such as a limitation value of 1.0 with a Most Limiting aggregation type or 0.0 with a Least Limiting aggregation type).

**Comments for aggregation method “weighted average”:**

Weighted Average:

For each mapunit, set its final rating value to the weighted average of all of its components' attribute values. If the components for the mapunit all have null attribute values, leave the rating null.

Comments for aggregation method “minimum or maximum”:

Minimum or Maximum:

For each mapunit, set its final rating to the attribute value for its component with the highest or lowest tiebreak value (depending upon the tiebreak rule parameter) with non-null tiebreak values taking precedence.

Comments for aggregation method “dominant component”:

Dominant Component:

For each mapunit, set its final rating to the attribute value for its component with the highest percent composition. If more than one component shares the high percent composition, use the highest or lowest non-null tiebreak value (depending upon the tiebreak rule parameter) as a tie breaker. Note that there will always be at least one non-null tiebreak value.

Comments for aggregation method “dominant condition”:

Dominant Condition:

For each mapunit, set its final rating to the attribute value associated with the condition that has the highest total percent composition from among all components associated with that mapunit. A condition is defined using the attribute value and primary and secondary constraint values. If more than one condition shares the high total percent composition, use the condition with the highest or lowest tiebreak value (depending upon the tiebreak rule parameter) with non-null tiebreak values taking precedence.

Attribute values are grouped by temporarily adding a copy of the Component DataTable (Groups) to the original DataSet with a child relationship to the Component DataTable. Rows within the parent Component table are deleted until only rows with unique attribute value/primary constraint/

secondary constraint triplets remain for each mapunit. Two new expression columns are then added to this table to calculate the sum of the percent compositions and the highest or lowest tiebreak value for each set of child records for each parent row. Once the setup is done, the mapunits are traversed and for each, the component records with the maximum total percent composition are returned and sorted appropriately by their highest or lowest tiebreak values.

**Our case:**  
**Comments for aggregation method “weighted average”:**

Weighted Average:

For each mapunit, set its final rating value to the weighted average of all of its components' attribute values. If the components for the mapunit all have null attribute values, leave the rating null.

The code executed in our case is the following.

There is a method used below on the data table “Comps” called “Compute”. An example is “dtC.Compute()”. A description of this method follows.

public object Compute (string expression, string filter);  
Computes the given expression on the current rows that pass the filter criteria. The expression parameter requires an aggregate function like Sum used below. The second parameter, filter, determines which rows are used in the expression.

case "weighted average":

The code first adds the column “totalcomppct” to the “Comps” data table “dtC”. It then loops through all the rows in the Comps data table (there are 2 rows) and assigns a value to this new “totalcomppct” column. The value is the aggregate sum for column “comppct\_r” where mukey=’64274’ and column “attributevalue” is not null. We know that the two rows have values of 5 and 85 for the “comppct\_r” column, so the sum is 90. In SQL readable form this translates to:

SELECT SUM(comppct\_r) FROM #sdvcomponents   
WHERE mukey=’64274’ and attributevalue is not null

IN SSMS, I entered the following:

SELECT SUM(comppct\_r) As SUMcomppct\_r FROM #sdvcomponents with(nolock)

WHERE mukey='64274' and attributevalue is not null;

GO

The result was 90, as expected:



dtC.Columns.Add("totalcomppct", Type.GetType("System.Int32"));

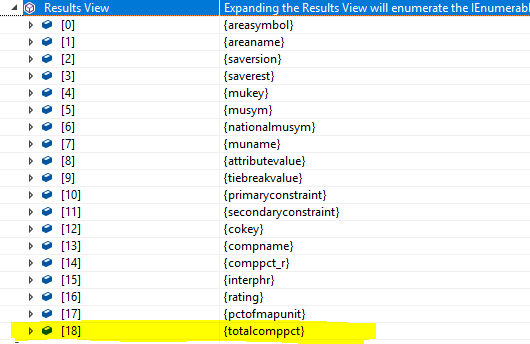
foreach (DataRow c in dtC.Rows)

{

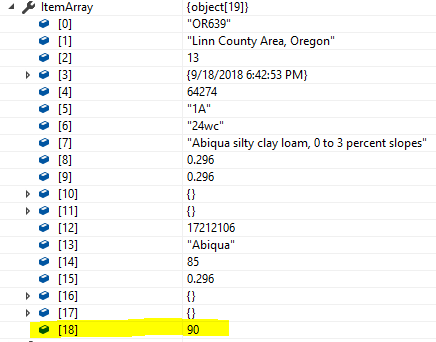
c["totalcomppct"] = dtC.Compute("sum(comppct\_r)", "mukey='" + c["mukey"].ToString() + "' and attributevalue is not null");

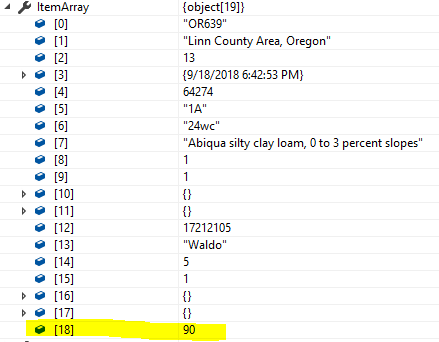
}

Adding the new column:



The result of running the SUM() aggregate function against each row in the “Comps” data table “dtC” is as follows.





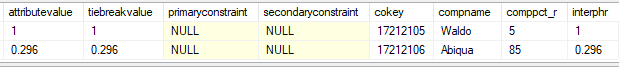
Next, two more columns are added named “doubleattribute” and “weightedattribute”. The doubleattribute column is assigned with the attributevalue column, and the weightedattribute column is assigned with the value of doubleattribute (attributevalue) X comppct\_r DIV totalcomppct.

dtC.Columns.Add("doubleattribute", Type.GetType("System.Double"), "attributevalue");

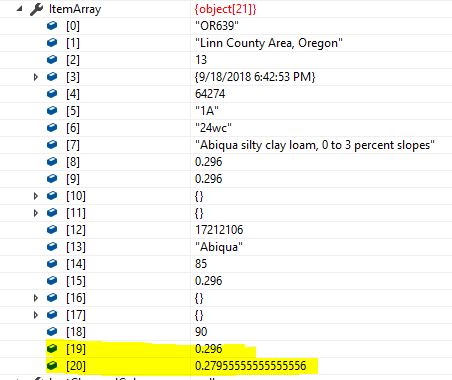
dtC.Columns.Add("weightedattribute", Type.GetType("System.Double"), "doubleattribute \* comppct\_r / totalcomppct");

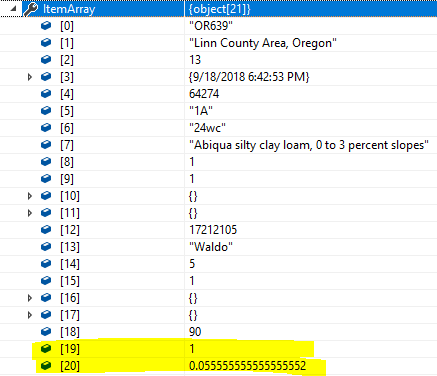
For the “doubleattribute” column, we already know that the “attributevalue” column has a value of 1 for the record with cokey=17212105 and a value of 0.296 for the record with cokey=17212106. For the “weightedattribute” column for the record with cokey=17212105, the equation to solve is:

1 X 5 DIV 90 = 5 / 90 which is 0.0555555555555552, and for the record with cokey=17117106, the equation to solve is: 0.296 X 85 DIV 90 = 25.16 / 90 which is 0.2795555555555556.



The result of adding the two columns above to the “Comps” data table “dtC” is as follows.





Next, we loop through the Mapunits data table dtM, which we know only had one row. The columns “rating” and “pctofmapunit” are calculated here using the two rows from the Components data table dtC. The “rating” column value is the SUM(weightedattribute) WHERE mukey=’64274’ and attributevalue is not null. We just need to add up the two row values for “weightedattribute” which are 0.0555555555555552 and 0.2795555555555556 = 0.3351111111111108. The “pctofmapunit” value is the SUM(comppct\_r) WHERE mukey=’64274’ and attributevalue is not null. Here we need to add up the two row values for “comppct\_r” which are 5 and 85 = 90. Therefore, the “rating” value is 0.335 and the “pctofmapunit” value is 90.

foreach (DataRow m in dtM.Rows)

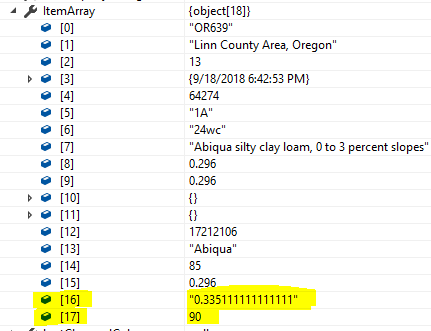
{

m["rating"] = dtC.Compute("sum(weightedattribute)", "mukey='" + m["mukey"].ToString() + "' and attributevalue is not null");

m["pctofmapunit"] = dtC.Compute("sum(comppct\_r)", "mukey='" + m["mukey"].ToString() + "' and attributevalue is not null");

}

The result of running the above for loop against the Mapunit data table “dtM” is as follows.



Next is the removal (cleanup) of the 3 temporary columns we just created for the Components data table “dtC”.

dtC.Columns.Remove("weightedattribute");

dtC.Columns.Remove("doubleattribute");

dtC.Columns.Remove("totalcomppct");

break;

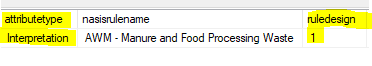
Next is the removal of all the columns at the component level that are no longer needed in the Results table (dtM).   
  
 dtM.Columns.Remove("cokey");

dtM.Columns.Remove("compname");

dtM.Columns.Remove("comppct\_r");

if (GetAttributeValue(DRRule, "attributetype").ToLower() == "interpretation" && (GetAttributeValue(DRRule, "ruledesign") == "1" || GetAttributeValue(DRRule, "ruledesign") == "2"))

dtM.Columns.Remove("interphr");

All these removal lines are executed, removing the cokey, compname, comppct\_r and interphr from the Mapunits data table “dtM” which is also the “Results” table. Notice the “if” statement above. This ends up being true in our case because “attributetype” is “interpretation” and the “ruledesign” is 1. Because of this, we also remove the column “interphr”.  
  


Next, we remove all of the columns that are no longer needed in the Results table, accept all changes and return.

DSData.Tables["Results"].Columns.Remove("attributevalue");

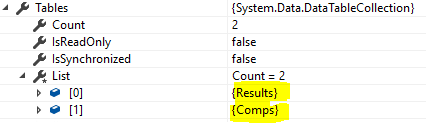
DSData.Tables["Results"].Columns.Remove("tiebreakvalue");

DSData.Tables["Results"].Columns.Remove("primaryconstraint");

DSData.Tables["Results"].Columns.Remove("secondaryconstraint");

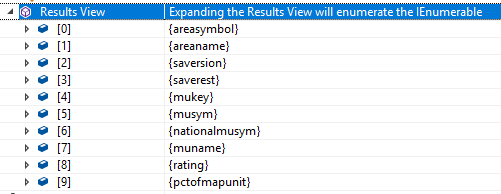
DSData.Tables["Results"].AcceptChanges();

DSData above is a DataSet and contains two tables: “Results” and “Comps” (Components). Recall that “Results” has one row of data and “Comps” has two rows of data.

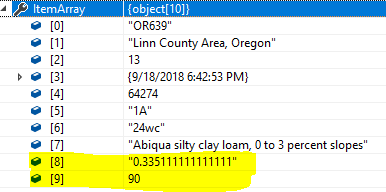


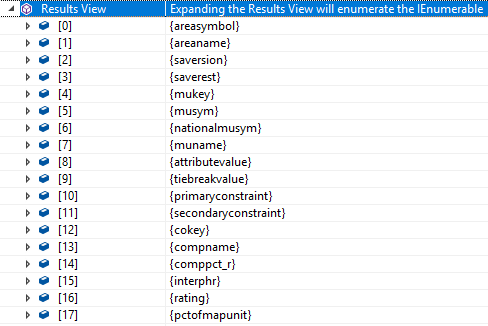
The following is what the “Results” table looks like at the end of this method.

The columns in the Results table are as follows:

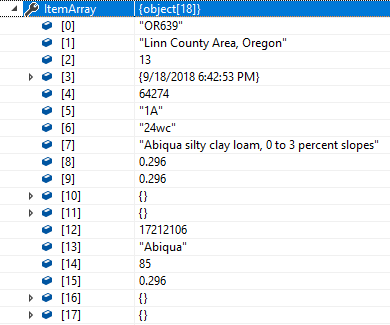


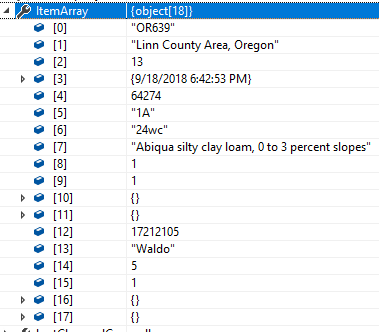
The Results table has one row of data and is as follows.



The columns in the Comps (Components) table are as follows.  


The Comps (Components) table has two rows of data and are as follows.





The last method SDVEngine calls is FinishForDisplay().

ds = FinishForDisplay(ds, db, DBModel, DRRule, AP, mukeys, distmdID);

FinishForDisplay() - This private method cleans up the results that are to be returned. First, if all mapunit ratings are null, it returns null since no ratings were calculated for any of the mapunits that were passed in. The Components table used in aggregation is removed. If a non-class soil interpretation was aggregated, then a new Components and Reasons tables are added. These two tables contain additional information for reporting purposes about how a mapunit rating was arrived at.

TODO: Edit the following if needed:  
- Results (areasymbol areaname saversion saverest mukey musym nationalmusym muname rating pctofmapunit)

The following tables are also in the returned DataSet for non-class soil interpretations only:

- Components (mukey cokey compname comppct\_r interphr)

with relationship ResultComponents to Results.

- Reasons (cokey interphrc interphr)

with relationship ComponentReasons to Components.

The first thing the FinishForDisplay() code does is make a list of mapunit keys and place them in the “mulist” string. Since we only have one mapunit in our case, the variable mulist is "'64274'". Next, we want query fields that are specific to the data model we are using. In our case, we are using the SDM or Soil Data Mart database model. The code has different values for the following variables depending upon the data model (MS Access, Soil Data Mart, NASIS, or Staging Server). I will only copy the Soil Data Mart code here. It is interesting that the default data model case is set to “Soil Data Mart” (see “default:” in the code below).

string mapunitkey, componentkey, interpstable, interpsfkey, from, where;

switch (DBModel)

{  
…

…

case SDVDatabaseModel.SoilDataMart:

default:

interpstable = "cointerp";

mapunitkey = "mapunit.mukey";

componentkey = "component.cokey";

interpsfkey = "cokey";

from = " from mapunit, component, cointerp";

where = " where mapunit.mukey=component.mukey and component.cokey=cointerp.cokey";

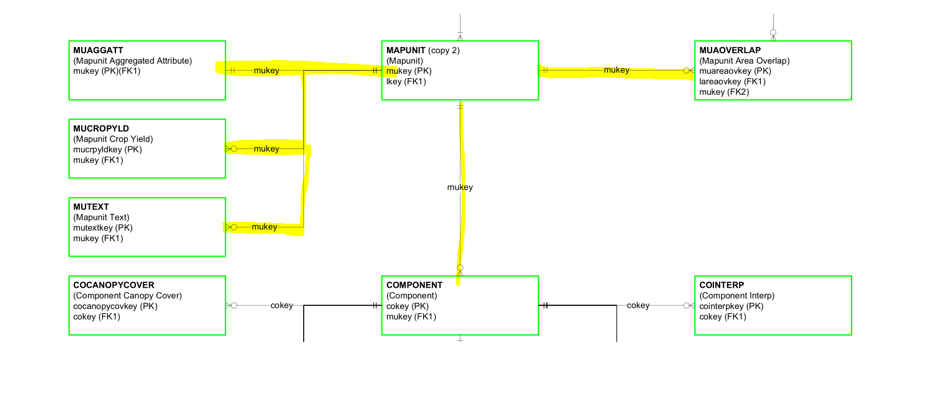
break;

…

…

}

Recall that the mapunit, component and cointerp tables are connected. The mapunit and component tables are linked using the “mukey” field and the component and cointerp tables are linked using the “cokey” field as shown in the diagram below.

  
  
Next, if a Components table exists, remove it. This is true for our case, since we have the “Comps” (Components) table in our DSData DataSet (recall we have “Results” and “Comps” tables in this data set).   
  
if (DSData.Tables.Contains("Comps")) {

DSData.Relations.Remove("MapunitComps");

DSData.Tables.Remove("Comps");

DSData.AcceptChanges();

}   
  
The if statement above is true, so in our DSData DataSet, we remove the “MapunitComps” relations (relationship table for Mapunit and Components which in our case was not needed anyway), then the “Comps” data table and finally we accept the changes (commit changes). You may ask why would we delete the Components table? It is because we are going to create a new one.

If the attribute that was aggregated is a soil interpretation, create and fill a new Components table. Notice that in our case the if statement directly below is true. Our attributetype is “interpretation” and we are not using the NASIS data model (we are using the Soil Data Mart one). The code “!=” is read as “not equals”.   
  
if (GetAttributeValue(DRRule, "attributetype").ToLower() == "interpretation"

&& DBModel != SDVDatabaseModel.NASIS)

{

Start by generating and running a query to retrieve all components back for the mapunits. Select the interphrc column temporarily to be used in cleaning up the records that aren't desired and the localphase column temporarily to modify the compnames on all remaining records.

First, we build a SQL SELECT query string using the variable “s”. After the following code is executed, “s” will be:

select mapunit.mukey as mukey, component.cokey as cokey, component.compname, component.comppct\_r, interphr, interphrc, component.localphase from mapunit, component, cointerp where mapunit.mukey=component.mukey and component.cokey=cointerp.cokey

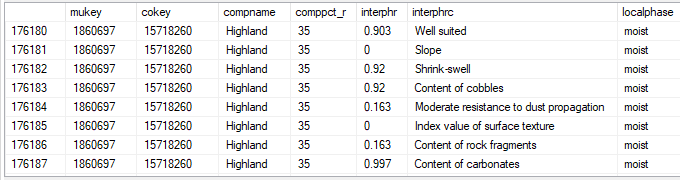
s = new StringBuilder("");

s.Append("select " + mapunitkey + " as mukey, " + componentkey +

" as cokey, component.compname, component.comppct\_r, interphr, interphrc,

component.localphase " + from + where);

The following are some records produced as a result of running the SELECT query directly above.



Three if-statements are next. We do have a mulist with length greater than zero (the variable mulist is "'64274'"), so the first if-statement is true. The second if-statement is false (in our case, the method FetchAllComps() returns false as described above), so we do append. The third if-statement is not executed because AP.ComponentPercentCutoff.Length = 0 in our case.

if (mulist.Length > 0)

s.Append(" and " + mapunitkey + " in (").Append(mulist).Append(")");

if (!FetchAllComps(AP))

s.Append(" and component.comppct\_r is not null");

if (AP.ComponentPercentCutoff.Length > 0)

s.Append(" and component.comppct\_r >= ").Append(AP.ComponentPercentCutoff);

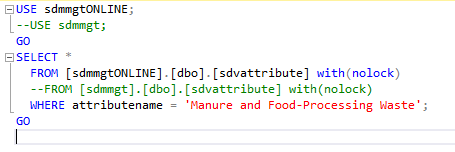
After the three if-statements are executed above, the variable “s” is now:

select mapunit.mukey as mukey, component.cokey as cokey, component.compname, component.comppct\_r, interphr, interphrc, component.localphase from mapunit, component, cointerp where mapunit.mukey=component.mukey and component.cokey=cointerp.cokey and mapunit.mukey in (‘64274’) and component.comppct\_r is not null

Next, we do more appending. The “Append” method retrieves the “nasisrulename” from the sdvattribute table in the sdmmgtONLINE database. In our case, “nasisrulename” value is “AWM – Manure and Food Processing Waste”.

s.Append(" and mrulename = '").Append(GetAttributeValue(DRRule,  
"nasisrulename")).Append("' and ruledepth = 0 order by " + mapunitkey + ",

component.comppct\_r desc");





After the append statement is executed, the variable “s” is now:

select mapunit.mukey as mukey, component.cokey as cokey, component.compname, component.comppct\_r, interphr, interphrc, component.localphase from mapunit, component, cointerp where mapunit.mukey=component.mukey and component.cokey=cointerp.cokey and mapunit.mukey in (‘64274’) and component.comppct\_r is not null and mrulename = ‘AWM – Manure and Food Processing Waste’ and ruledepth = 0 order by mapunit.mukey, component.comppct\_r desc

The next statement adds the “Components” data table to the DSData dataset by running the SELECT statement directly above. Note that the “mrulename” and “ruledepth” fields above are from the “cointerp” table. Two rows of data are returned, so the new “Components” data table has 2 rows of data right now.

DSData.Tables.Add(SQLToDataTable(db, DBModel, s.ToString(), "Components"));



Next, we are creating a “data relations” table called “MapunitComponents” between our “Results” table and our current “Components” table using the “mukey” field to link the two tables, and adding it to the main DataSet, DSData. This means the main DataSet, DSData, now consists of the “Results” and “Components” data tables and the “MapunitComponents” data relation. The parent table for this relation is the “Mapunit” (Results) data table and the child table for this relation is the “Components” data table. The common column connecting these two data tables is “mukey” (mapunit key). The column “mukey” serves as the link between the two data tables. The last parameter is “false” which is a value that indicates whether to create constraints (in our case we are specifying not to create constraints). A DataRelation also has a Nested property which, when set to true in our case, causes the rows from the child table to be nested within the associated row from the parent table when written as XML elements using WriteXml .

DSData.Relations.Add("MapunitComponents", DSData.Tables["Results"].Columns["mukey"], DSData.Tables["Components"].Columns["mukey"], false);

DSData.Relations["MapunitComponents"].Nested = true;

Traverse the list of mapunits and delete all components that don't have the same interpretation rating class as the final rating that was calculated for the associated mapunit. Then traverse all remaining components and alter their compnames using the localphase as needed. Remove the two temporary columns.

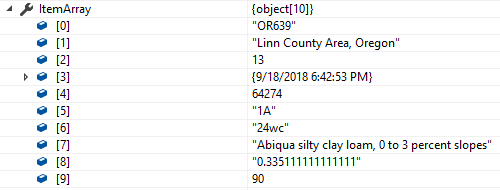
We start out by defining the Mapunit data table (dtM) and Components data table (dtC). Note that the Mapunit data table is taken from the Results data table and the Components data table is taken from the Components data table.

DataTable dtM = DSData.Tables["Results"];

DataTable dtC = DSData.Tables["Components"];

DataRow[] drs;

The Mapunit data table (dtM) has one row of data and is the following:



The Components data table (dtC) has two rows of data and is the following:



Next, the string “q” variable is assigned "'". This is because the sdvattribute.attributelogicaldatatype’s value is “string”.

q = GetAttributeValue(DRRule, "attributelogicaldatatype").ToLower() == "integer" || GetAttributeValue(DRRule, "attributelogicaldatatype").ToLower() == "float" ? "" : "'";

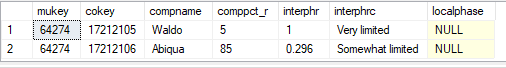


Next, we go through the Mapunit data table, dtM, which has one row and setup the “sel” string variable. We run that “sel” string (a SQL SELECT statement) and then go through each of those rows and perform row deletions. After that we accept or commit the changes.

The variable “sel” will be equal to:

SELECT \* FROM #components --[data table Components]   
WHERE mukey='64274' and (interphrc is null or interphrc<> '0.335111111111111')

This returns the same two rows of data.



Now we need to delete these two component rows of data since they don't have the same interpretation rating class as the final rating that was calculated for the associated mapunit (0.335111111111111). This means the Components data table now has 0 rows of data.



foreach (DataRow r in dtM.Rows) {

string sel = "mukey='" + r["mukey"].ToString() + "' and ";

if (DBNull.Value == r["rating"])

sel += "interphrc is not null";

else

sel += "(interphrc is null or interphrc<>" + q + r["rating"].ToString().Replace("'",

"''") + q + ")";

drs = dtC.Select(sel);

foreach (DataRow c in drs) c.Delete();

}

dtC.AcceptChanges();

Next, if we had Components rows of data (we have 0 rows, so this next part for loop is not executed) and if the localphase value was not NULL, we change the “compname” to the compname value + “, “ + the localphase value. For example, for the “Waldo” compname with a localphase of “cool”, then the new compname would be “Waldo, cool”. Again, this is just an example and is not what our data returns. At the end of this part the localphase and intephrc columns are removed from the data table “Components”.

Our “Components” table before the following code is executed:



foreach (DataRow c in dtC.Rows) {

if (c["localphase"] != System.DBNull.Value) c["compname"] = c["compname"].ToString() +

", " + c["localphase"].ToString();

}

dtC.Columns.Remove("localphase");

dtC.Columns.Remove("interphrc");

dtC.AcceptChanges();

Our “Components” after the above code is executed:



If the attribute that was aggregated is a non-class soil interpretation, create and fill a new Reasons table. In our case ruledesign is “1”, but we have no rows of data in our “Components” data table, so the code after the statement “if (s.Length > 0)” below is not executed. This is because the variable “s” in our case is the empty string and has a length of zero.

if (GetAttributeValue(DRRule, "ruledesign") == "1" || GetAttributeValue(DRRule, "ruledesign") == "2") {

s = new StringBuilder("");

foreach (DataRow c in dtC.Rows) s.Append("'").Append(c["cokey"].ToString()).Append("'");

**if (s.Length > 0) {**

q = "select " + interpsfkey + " as cokey, interphrc, interphr from " + interpstable +

" c where mrulename = '" + GetAttributeValue(DRRule, "nasisrulename") +

"' and ruledepth = 1 and " + interpsfkey + " in (" + s.Replace("''",

"','").ToString() + ") and exists (select \* from " + interpstable + " where

mrulename = c.mrulename and " + interpsfkey + " = c." + interpsfkey + " and

ruledepth = 0 and interphr is not null) " + "order by seqnum";

DSData.Tables.Add(SQLToDataTable(db, DBModel, q, "Reasons"));

DSData.Relations.Add("ComponentReasons", dtC.Columns["cokey"],

DSData.Tables["Reasons"].Columns["cokey"], false);

DSData.Relations["ComponentReasons"].Nested = true;

}

}

If the attribute being aggregated returns float values, then round those values to the precision that they should have. If they are supposed to be integer values, then round them back to integers. In our case, the “effectivelogicaldatatype” value is “string”, the “AP.AggregationMethod” value is “weighted average”, the “attributetype is “interpretation”, and the “attributeprecision” value is NULL.

The “precision” string variable below will be equal to “3” because we have an attributetype of interpretation and an aggregation method of “weighted average”. We then go through each row in our “Results” data table (we have one row), and change the “rating” value to have a precision of 3. This means the rating of 0.335111111111111 is changed to 0.335.

System.Globalization.NumberFormatInfo nfi = new System.Globalization.CultureInfo("en-US", false).NumberFormat;

nfi.NumberGroupSeparator = "";

bool isint = (GetAttributeValue(DRRule, "effectivelogicaldatatype").ToLower() == "integer");

bool isWtaInterp = (AP.AggregationMethod == "weighted average" && GetAttributeValue(DRRule, "attributetype").ToLower() == "interpretation");

if (isint || isWtaInterp || GetAttributeValue(DRRule, "effectivelogicaldatatype").ToLower() == "float") {

DSData.AcceptChanges();

string precision = isint ? "0" : isWtaInterp ? "3" : GetAttributeValue(DRRule,

"attributeprecision");

nfi.NumberDecimalDigits = Convert.ToInt32(precision);

foreach (DataRow r in DSData.Tables["Results"].Rows)

if (r["rating"] != System.DBNull.Value)

r["rating"] = (Math.Round(Convert.ToDouble(r["rating"]), nfi.NumberDecimalDigits,

MidpointRounding.AwayFromZero)).ToString("N", nfi);

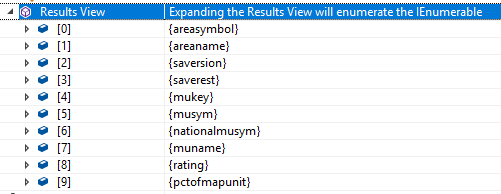
}

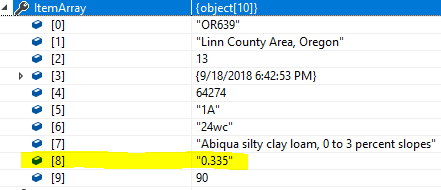
Accept all changes and return. We commit all changes to the DSData data set and return it.

DSData.AcceptChanges();

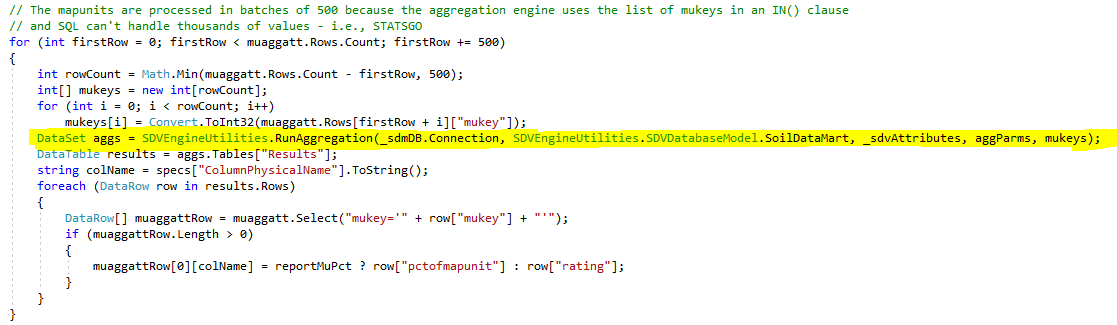
return DSData;

This is our “Results” data table:





We are done with the SDVEngine side of the code in the SDVEngine solution. We return execution back to the SoilDataManagerService solution.



After RunAggregation() is done with its work (highlighted in yellow above), it returns a DataSet to the method DeriveAggAtt(), and the code in the DeriveAggAtts() method runs. This code is shown above and below. The “Results” DataTable retrieved from RunAggregation() is returned and used for the rest of the processing.

The following code is in the public void DeriveAggAtts(DataTable muaggatt) method in the MapunitAggregatedAttributes.cs file in the SoilDataManagerService solution.

In our case, the specs[“ColumnPhysicalName”] is “awmmfpwwta”. We loop through the one row in our Results data table. Recall that we had a “rating” of 0.335 and a “pctofmapunit” of 90. In our case “reportMuPct” = “false” because the “reportmupercent” in our “Specs” table is 0.





So what gets executed below?

muaggattRow[0][ awmmfpwwta] = row[“rating”] = 0.335.

This means the “awmmfpwwta” column in the “muaggatt” table’s first row gets set to 0.335 (it had NULL).



 🡨This column gets set to 0.335.

DataTable results = aggs.Tables["Results"];

string colName = specs["ColumnPhysicalName"].ToString();

foreach (DataRow row in results.Rows)

{

DataRow[] muaggattRow = muaggatt.Select("mukey='" + row["mukey"] + "'");

if (muaggattRow.Length > 0)

{

muaggattRow[0][colName] = reportMuPct ? row["pctofmapunit"] : row["rating"];

}

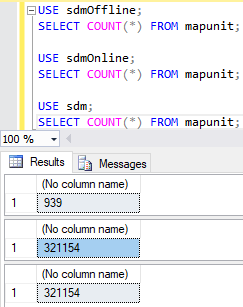
}

Currently, when using the “sdmOffline” database, the “muaggatt” table is joined with the “mapunit” table, producing 939 rows. The “muaggatt” table has 36 columns that need populated. What I just explained in this document was just one column in the “muaggatt” table (awmmfpwwta) getting populated using one mapunit (mapunit ‘64274’). Note that the SDVEngine is called with batches of 500 mapunits at one time. This means the column I used, “awmmfpwwta”, would be processed with 500 mapunits in each call to SDVEngine until all 939 rows were processed. So just for column “awmmfpwwta”, SDVEngine would need to be called 939 / 500 times = 2 times. We currently need to fill in 36 columns of muaggatt data, so that means SDVEngine is currently being called 36 x 2 = 72 times.

If we add 124 more rows to the muaggatt table, that means we need (124 + 36) x 2 calls to the SDVEngine or 160 x 2 = 320 times (the number of calls jumps from 72 times to 320 times). This is 4.4 times as many calls.

Notice that the more mapunit rows you have, the more processing will be needed as there will be more calls to the SDVEngine. For example, for the “sdm” snapshot database, table “mapunit” has 321,154 rows of data. This means for each of the current 36 muaggatt columns, the SDVEngine would be called 321154 / 500 = 643 times. We currently have 36 columns, so that means the SDVEngine would be called 36 x 643 = 23,148 times, a significant jump. I checked the code and it is using the “sdmOffline” database thankfully, so the mapunit table total row count should be 939 rows. I wanted to show that as you increase the number of mapunits, the processing time can be significant.

There are 2 solutions that I had to open in Visual Studio in order to document this. One is the SoilDataManagerService and the other is the SDVEngine. Note that the SoilDataManagerService is using the “sdmOffline” database and the SDVEngine is using the “sdm” snapshot database. I do not know why this is different. I performed 3 quick queries to see the number of rows in each database for table “mapunit” and found a difference between sdmOffline and sdm as shown below.



There was also a big difference for the “cointerp” table. While sdmOffline had 758,545 rows, the sdm database returned 537,720,785 rows (537+ million).

