



FME Desktop[®]

**Database (SQL Server)
Pathway Training**

FME 2014-SP3 Edition



Safe Software Inc. makes no warranty either expressed or implied, including, but not limited to, any implied warranties of merchantability or fitness for a particular purpose regarding these materials, and makes such materials available solely on an "as-is" basis.

In no event shall Safe Software Inc. be liable to anyone for special, collateral, incidental, or consequential damages in connection with or arising out of purchase or use of these materials. The sole and exclusive liability of Safe Software Inc., regardless of the form or action, shall not exceed the purchase price of the materials described herein.

This manual describes the functionality and use of the software at the time of publication. The software described herein, and the descriptions themselves, are subject to change without notice.

Copyright

© 1994 – 2014 Safe Software Inc. All rights are reserved.

Revisions

Every effort has been made to ensure the accuracy of this document. Safe Software Inc. regrets any errors and omissions that may occur and would appreciate being informed of any errors found. Safe Software Inc. will correct any such errors and omissions in a subsequent version, as feasible. Please contact us at:

Safe Software Inc.
Suite 2017, 7445 – 132nd Street
Surrey, BC
Canada
V3W1J8

www.safe.com

Safe Software Inc. assumes no responsibility for any errors in this document or their consequences, and reserves the right to make improvements and changes to this document without notice.

Trademarks

FME is a registered trademark of Safe Software Inc.

All brand or product names mentioned herein may be trademarks or registered trademarks of their respective holders and should be noted as such.

Documentation Information

Document Name: FME Desktop Database Pathway Training Manual
FME Version: FME 2014-SP3 32-bit
Operating System: Windows 7 SP-1, 64-bit
Database: Microsoft SQL Server Express 2012, 64-bit
Updated: September 2014

Introduction.....	5
Database Pathway	5
FME Version.....	5
Sample Data.....	5
Supported Database.....	5
Connecting to a Spatial Database	6
Basic Connection Parameters	6
Connecting to SQL Server.....	6
Creative Feature Reading	10
Where Clause.....	10
Search Envelope	11
Concatenated Parameters.....	14
FeatureReader	18
Updating Features	22
Controlling Translations.....	22
Writer Parameters	23
Feature Type Parameters.....	24
Format Attributes	25
Parameter Priority.....	26
Time and Date Attributes in Spatial Databases.....	37
Formatting Date Attributes with Transformers.....	37
Coordinate System Granularity	38
Supported Formats.....	38
Multiple Geometries	39
Multiple Geometry Writing	39
Multiple Geometry Reading	40
Database Transformers.....	48
SQLExecutor	48
SQLCreator	48
Geometry.....	52
Supported Geometries	52
Spatial Indices	52
Geometry or Geography?.....	53
Cleaning Geometry.....	53
Performance	54
Default Performance.....	54
Transaction Interval	54
Bulk Insert.....	55
Start Transaction At.....	55
Session Review	56
What You Should Have Learned from this Session	56

Introduction



This training material is part of the FME Training Pathway system.

Database Pathway

This training material is part of the FME Training Database Pathway.

It contains advanced content and assumes that the user is familiar with all of the concepts and practices covered by the FME Database Pathway Tutorial, and the FME Desktop Basic Training Course.

FME Version

This training material is designed specifically for use with FME2014-SP3. You may not have some of the functionality described if you use an older version of FME.

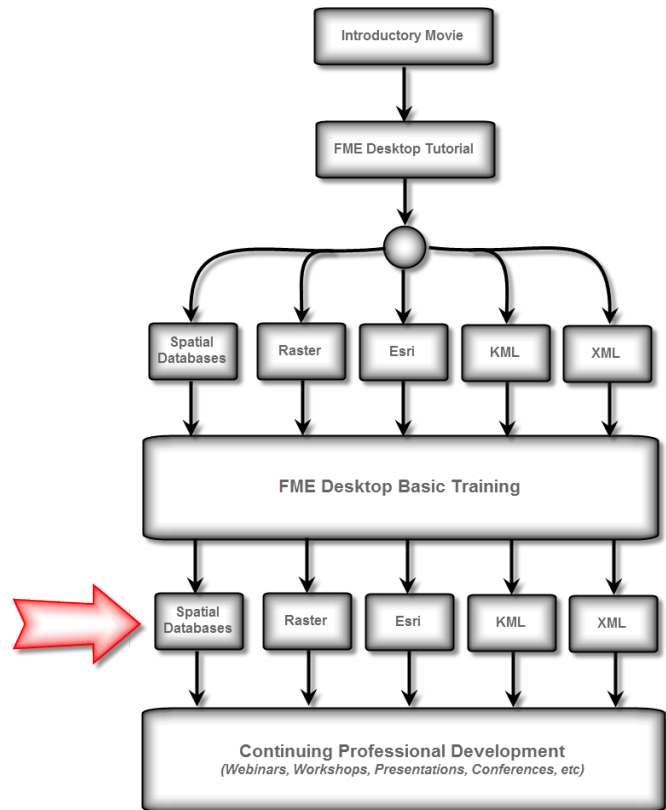
Sample Data

This sample data required to carry out the examples and exercises in this document can be obtained from:

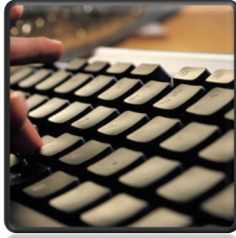
www.safe.com/fmedata

Supported Database

For the purposes of simplicity, this training includes documented steps for SQL Server 2012 only. In particular it was created using SQL Server Express 2012 SP1



Connecting to a Spatial Database



Connecting to the database is the one step all FME operations must perform.

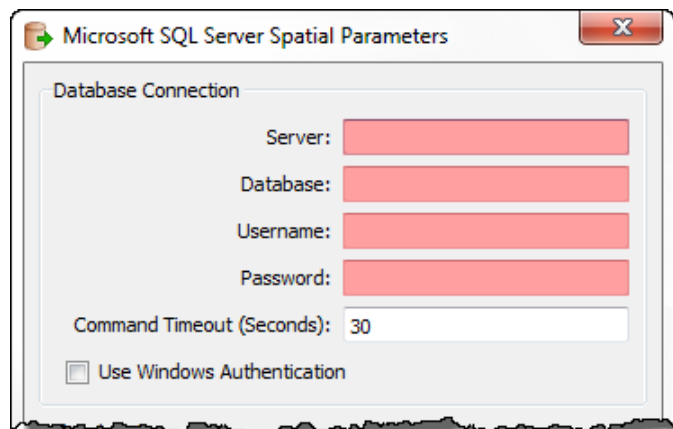
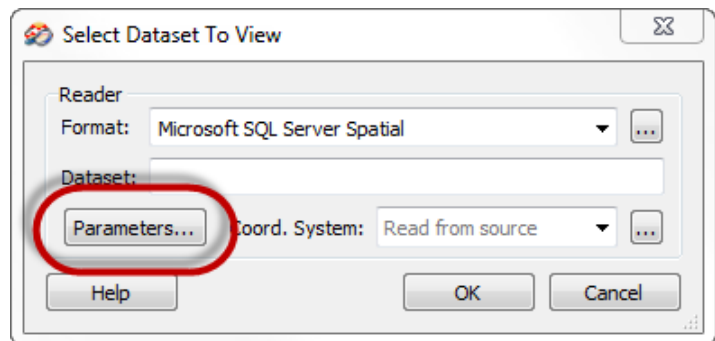
Connecting to a database is slightly different to selecting a file for a file/folder-based format. The operation relies much more on format specific parameters.

Basic Connection Parameters

The basic connection parameters are:

- Host Name
- Database Name
- Username
- Password
- Network Port Number

These parameters may differ slightly for each format, but will always be found in any Reader/Writer dialog by clicking on the “Parameters...” button.



Connecting to SQL Server

Connecting to a SQL Server database requires just four of the basic connection parameters. Port number is not required.

There is an additional option to use Windows Authentication.



Example 1: Connection Parameters	
Scenario	FME user; Planning Department
Data	Parks (MapInfo TAB)
Overall Goal	Test connection parameters by writing City Parks data to SQL Server
Demonstrates	Connection parameters and database writing
Finished Workspace	C:\FMEData2014\Workspaces\Database\mssql1-complete.fmw

Follow these steps to test the connection to your SQL Server database.

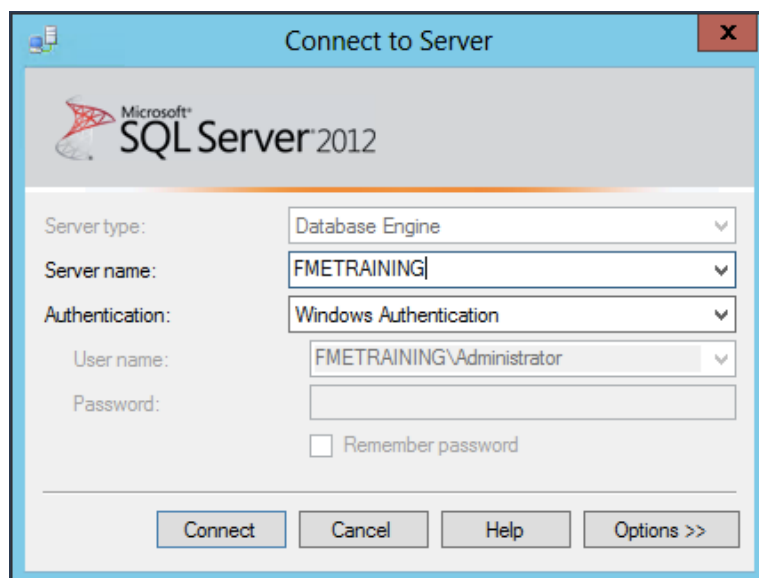
1) Start SQL Server Management Studio

Start the SQL Server Management Studio. It is found under *Start > All Programs > Microsoft SQL Server 2012 > SQL Server Management Studio*



When the management studio starts, it will prompt for a login.

The Server name field is important, as it provides the server name for FME to connect to. For this course, the server name is FMETRAINING (it is not case-sensitive).



If that fails, simply use a dot (period) character by itself.

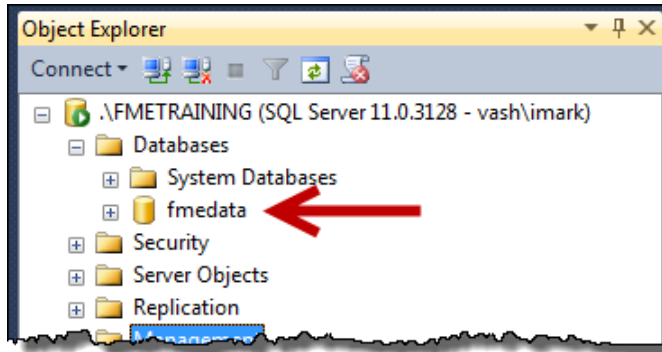
At the time of writing, the official FME training uses Windows Authentication to connect, so leave that setting as it is and click **Connect**.

3) Browse for Database

You will now be connected to the SQL Server database.

In the Object Explorer window, click the + button next to the Databases entry to list databases.

Select a suitable database for training use.



For FME training we recommend (and will illustrate examples) using a database called *fmedata*.

NB: If there is no database listed, then right-click on Databases, and choose New Database, in order to create one.

4) Start FME Workbench

Start FME Workbench, and open the Generate Workspace dialog.

Set up a translation as follows:

Reader Format MapInfo TAB (MITAB)
Reader Dataset C:\FMEData2014\Data\Parks\Parks.tab

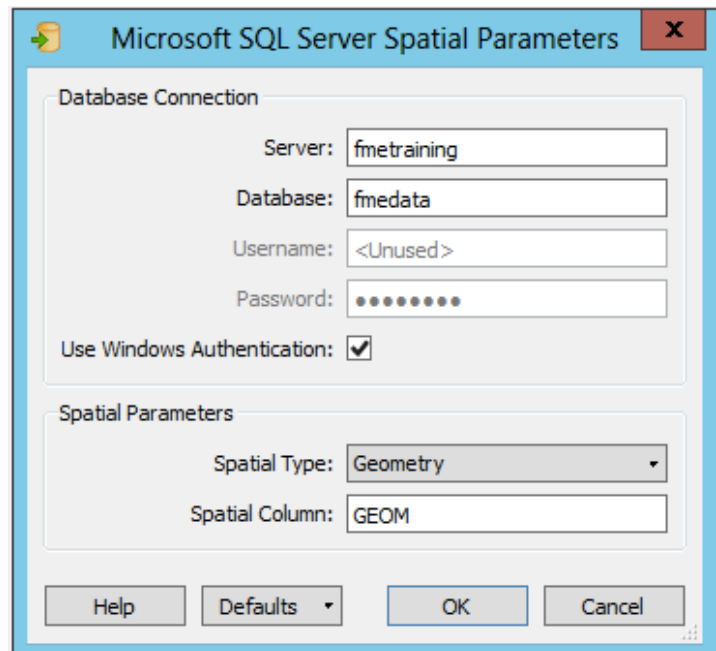
Writer Format Microsoft SQL Server Spatial

Click the writer parameters button. In the parameters dialog enter the database connection parameters.

In FME training this will usually be:

Server fmetraining
Database fmedata
Use Windows Authentication Yes

Click **OK**, and then **OK** again, to create the workspace.



5) Run Workspace

Run the workspace. The foot of the log file will report success as follows:

```
-----
Total Features Written                                     80
-----
-----
Features Read Summary
-----
Parks                                                     80
-----
Total Features Read                                       80
-----
-----
Features Written Summary
-----
Parks                                                     80
-----
Total Features Written                                     80
-----
Closing native MapInfo reader
Translation was SUCCESSFUL with 0 warning(s) (80 feature(s) output)
FME Session Duration: 1.9 seconds. (CPU: 0.5s user, 0.1s system)
END - ProcessID: 7632, peak process memory usage: 97264 kB, current process memc
Translation was SUCCESSFUL
```

If the log reports a message like one of these:

- SQL Server does not exist or access denied.
- Login failed for user 'xxxx'.

...then you should check the connection parameters entered into the dialog and, if necessary, re-check these against the parameters entered into the SQL Server Management Studio.

Creative Feature Reading



Rather than a plain reader, there are some quite creative ways by which database features can be read using Workbench.

Using FME to read from a database should be carefully planned and considered. Frequently not every feature in every table is required, and yet that is what a user might be doing.

The fewer features that are read from the database, the quicker the read will be, the less system resources are used, and the faster the overall translation will be.



Chef Bimm says...

'Think of a database like a restaurant. A sensible person would browse the menu, and order just the dishes that they want. A foolish person would order everything and waste the food they didn't really need.

Like a restaurant, it's very expensive and time-consuming to order all data from a database just to discard most of it. Far better to order only the data you intend to consume in the current session!'

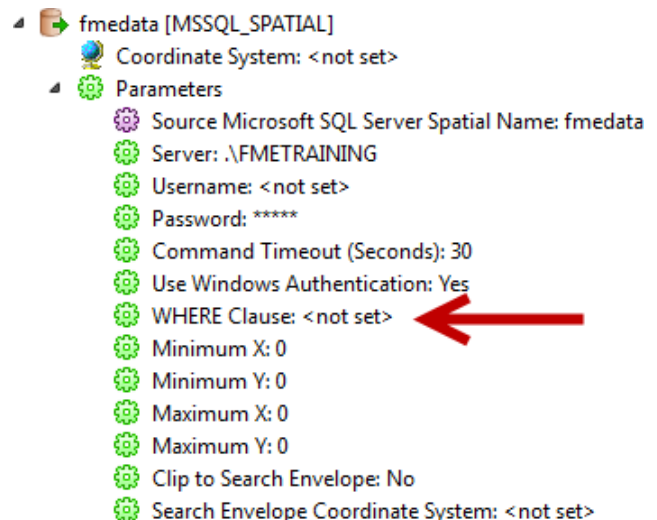
There are a number of items of functionality that can improve the performance of database reading in this way:

- Where Clause
- Search Envelope
- Concatenated Parameters
- FeatureReader

Where Clause

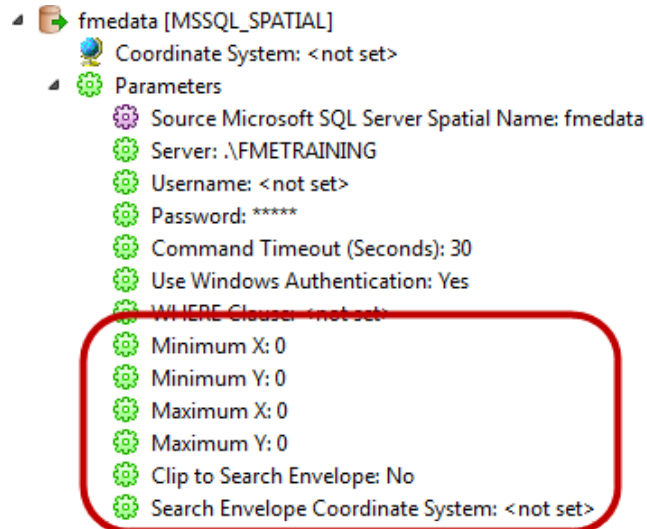
Most database readers will have a "where clause" parameter. Here a query can be set, so that only features that pass the query will be returned to FME.

This employs database query tools – which in turn make use of database indices – and is a lot more efficient than reading an entire table and then filtering it with a *Tester* transformer.



Search Envelope

Similar to a where clause, “search envelope” parameters set a spatial query; only features that fall inside the specified extents will be returned to FME.



Again, this employs native database functionality, and is more efficient than reading the entire table and then clipping it with a *Clipper* transformer.

An optional “Clip to Search Envelope” parameter defines whether features will be clipped where they cross the defined extents, or be allowed to pass completely where at least a part of them falls inside the extents.

Of course the limitation here is that the four parameters only define a rectangular envelope.



To use a search envelope requires a spatial index to exist for that table; FME can't instruct the database to do a spatial query without one. If a format doesn't support a spatial index being created on a view, then FME will not be able to do spatial queries on that view.

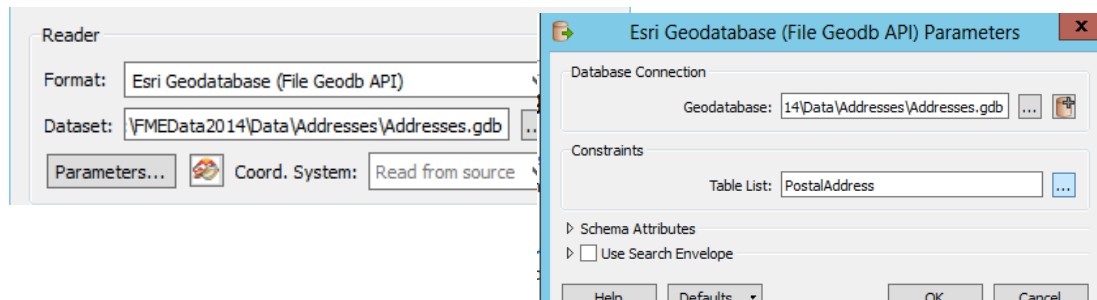


Example 2: Where Clause	
Scenario	FME user; Planning Department
Data	Postal Address Data (ESRI File Geodatabase)
Overall Goal	Read address data with a query
Demonstrates	Where Clause parameters
Finished Workspace	C:\FMEData2014\Workspaces\Database\mssql2-complete.fmw

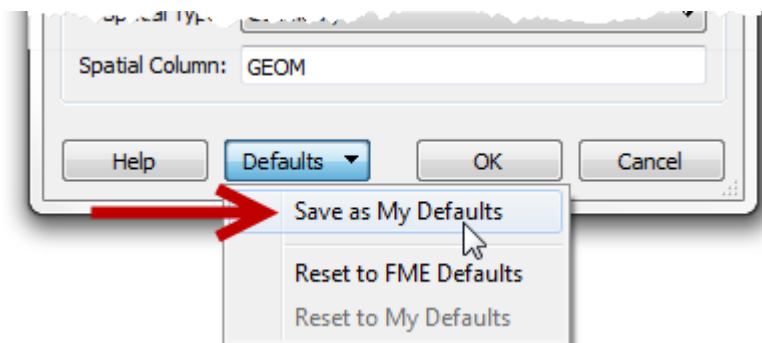
1) Start FME Workbench

Start Workbench if necessary and open the Generate Workspace dialog. Set up a translation as follows:

Reader Format Esri Geodatabase (File Geodb API)
Reader Dataset C:\FMEData2014\Data\Addresses\Addresses.gdb
Reader Parameter Table List: PostalAddress



Writer Format Microsoft SQL Server Spatial
Writer Parameters Enter the database connection parameters as before.



Once you are sure the connection parameters are correct, then choose the “Save as My Defaults” option at the foot of the parameters dialog.

This will save the parameters and prevent them having to be entered again and again.

4) Redirect to Inspector, Save and Run Workspace

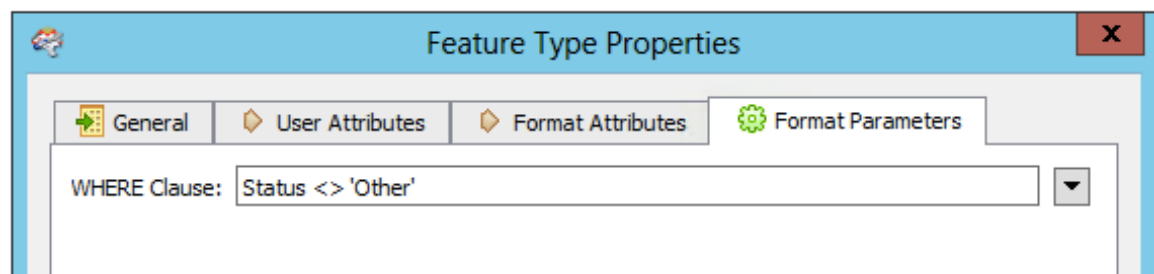
Turn on 'Redirect to Inspection Application'. Save and run the workspace. Note how many features were read and how long it took.

On my computer it reads 13,597 features in 8.6 seconds.

5) Set Where Clause

The Planning Department is not interested in PostalAddresses that have a Status of Other.

Set the PostalAddress reader feature type 'Format Parameters' WHERE Clause to be Status <> 'Other'. For some versions of SQL, 'not equal' is <>, for others it is !=.



6) Re-run Workspace

Re-run the workspace. Note how many features were read now and compare how long it took.

On my computer it's now 12,210 features in 7.1 seconds.



Concatenated Parameters

The problem with the where clause – as with other Reader/Writer parameters – is that it is difficult to get user input and apply it to the clause.

Simply publishing the parameter is not useful because the user would have to enter the full clause (<field> = <value>), when often only the <value> part is required as input.

This is where a concatenated parameter comes in. It is a parameter that is built of a constant string (the <field> part) and a user-defined value (the <value> part).

Remember that you'll still need to put whatever kind of quoting the database is expecting (for SQL Server this is single quotes) around the value part of the parameter.

NB: *Scripted Parameters do a similar task, but for more complex scenarios where the value has to be incorporated using a Python or Tcl script. This would be more useful for manipulating the search envelope parameter values.*



Example 3: Concatenated Parameter	
Scenario	FME user; Planning Department
Data	SQL Server (Postal Address Data)
Overall Goal	Read address data with a user-defined query
Demonstrates	Concatenated parameters for a database
Starting Workspace	C:\FMEData2014\Workspaces\Database\mssql3-begin.fmw
Finished Workspace	C:\FMEData2014\Workspaces\Database\mssql3a-complete.fmw C:\FMEData2014\Workspaces\Database\mssql3b-complete.fmw

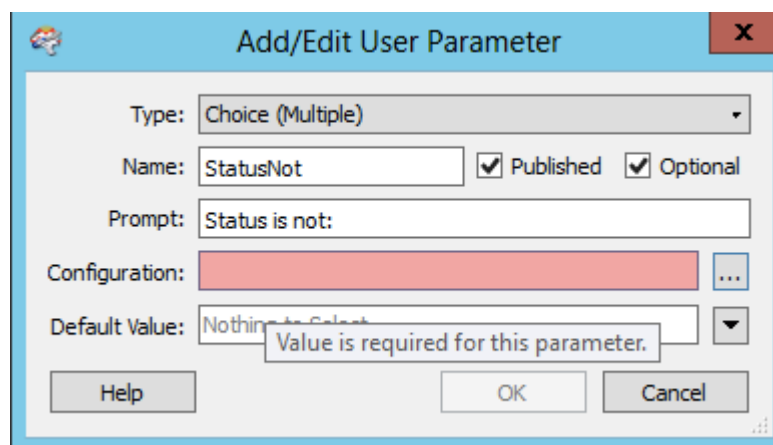
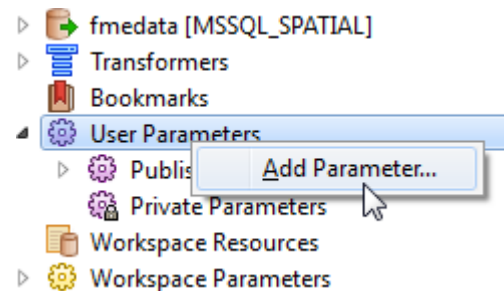
Here we wish to let the user choose which Status to read data from. We will want to create a WHERE Status NOT IN () statement. It will require the use of two parameters concatenated together.

1) Start FME Workbench

Start Workbench (if necessary) and open the workspace from the previous example. Alternatively you can open C:\FMEData2014\Workspaces\Database\mssql3-begin.fmw

2) Add Parameter

Right-click on "User Parameters" in the Navigator window and choose Add Parameter.

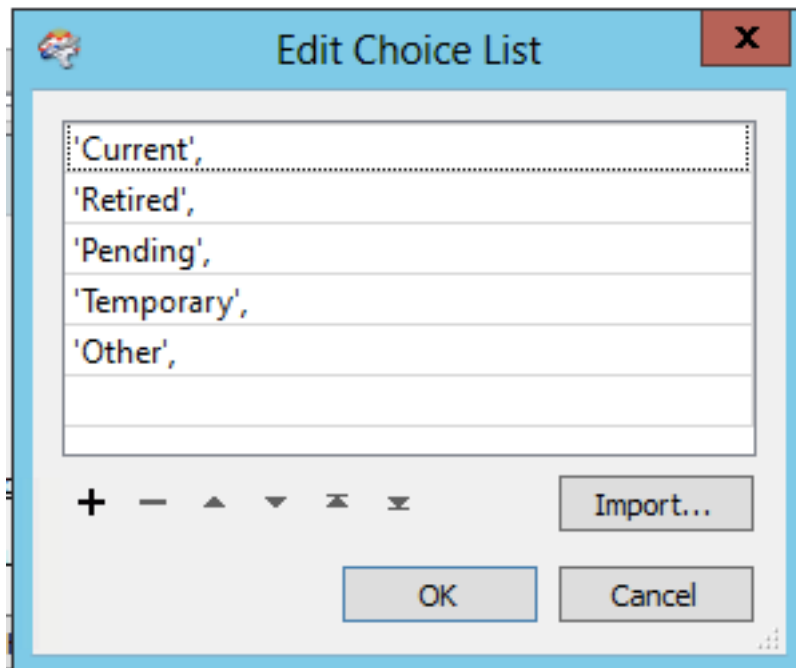


When prompted, choose a parameter of type 'Choice (Multiple)'. Set the name to StatusNot and the prompt to 'Status is not'.

Next, press the 'Configuration' button. The purpose of the entries here are to populate the NOT IN() clause in a later parameter. The NOT IN list is comma separated, and the text values must be in single quotes.

Add the following to the Choice List:

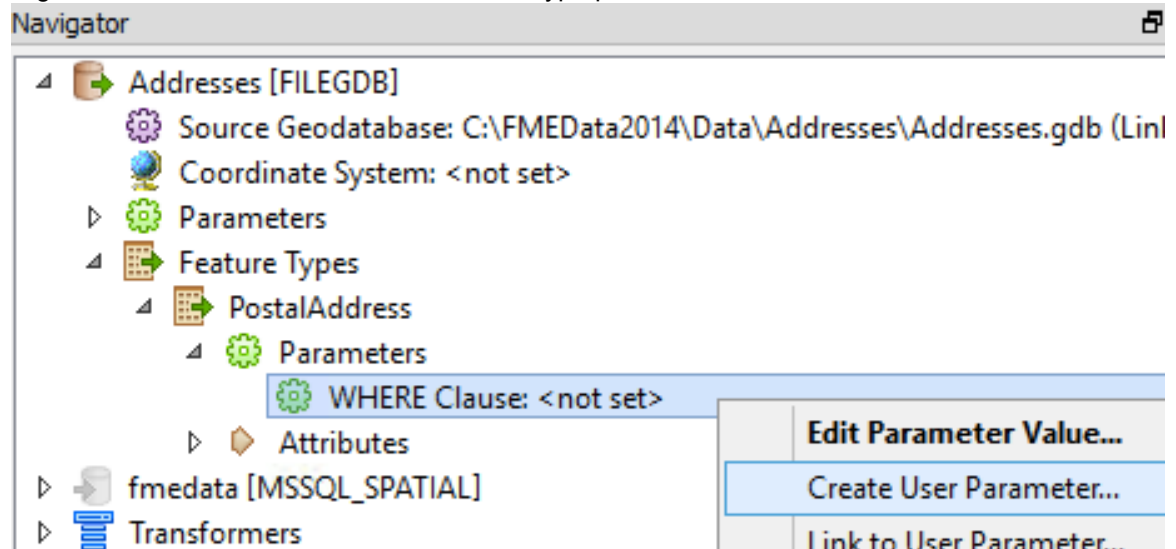
'Current',
'Retired',
'Pending',
'Temporary',
'Other',



Click OK to close the Edit Choice List, and OK again to close the Add/Edit Parameter.

3) Add Parameter

Right-click the Where Clause reader feature type parameter and choose Create User Parameter.

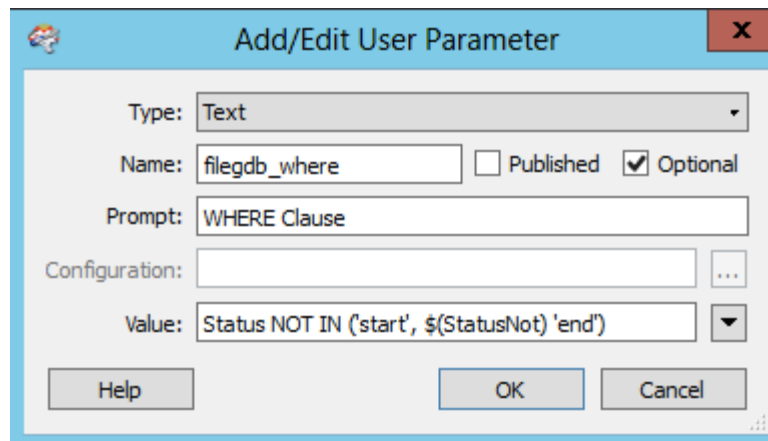


Leave the type as Text, and set the Where clause default value to:

Status NOT IN ('start', \$(StatusNot) 'end')

The 'start', and 'end' values are there as placeholders. This ensures that there are a correct number of commas for the SQL statement, and that the statement will still work if no Status values are selected.

The easiest method is to use the Text Editor where the "Status NOT IN" part can be typed manually (as a constant) and the published parameter can be selected from a list.



Turn off the Published flag, so the user is not prompted to set this concatenated parameter.

This is now concatenated with the previous parameter to form the required WHERE clause.

4) Run Workspace

It's time to load the database. Turn off 'Redirect to Inspection Application'.

Run the workspace using prompt and run.

When prompted, enter a Status of Other into the field provided.

Advanced Task

Although it's no longer database related, the Status selection dialog could be cleaned up by changing the Published Parameter type from Choice (Multiple) to Choice with Alias (Multiple).

FeatureReader

The *FeatureReader* transformer is one that acts – as the name suggests – as a reader in itself. The idea is that each incoming feature acts as a query to a database (or, in fact, any dataset) that can include both spatial and non-spatial components. This way queries can be carried out mid-translation, rather than through Reader parameters.

Incoming features are known as Initiators. Each of them causes a single query to be carried out through a reader. The query can use the geometry of the incoming feature as a base against which to test a spatial predicate, and the reader returns one or more features as the result of the query.

For example, an incoming line feature (maybe a road) can be used to define the base for an intersection query against linear database features such as rivers or rail.



Example 4: FeatureReader	
Scenario	FME user; Planning Department
Data	Zoning Data (MapInfo TAB) Postal Address Data (SQL Server)
Overall Goal	Read address data within a user-defined zoning area
Demonstrates	FeatureReader Transformer
Finished Workspace	C:\FMEData2014\Workspaces\Database\mssql4-complete.fmw

In this example we wish to let the user select a set of addresses by zoning category. However, zoning is not an attribute of the address data; therefore we will have to carry out an extract using the boundary of the zone in the FeatureReader transformer.

1) Start FME Workbench

Start Workbench if necessary and begin with an empty workspace.

2) Add Reader

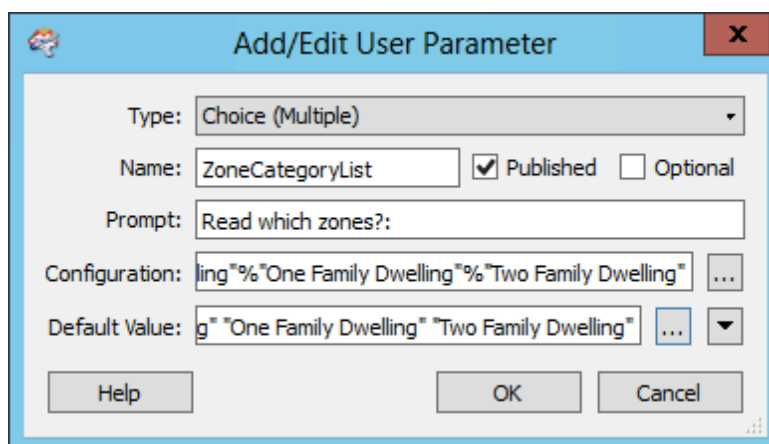
First let's add a reader (using Readers > Add Reader) to read the following dataset:

Reader Format MapInfo TAB (MITAB)
Reader Dataset C:\FMEData2014\Data\Zoning\Zones.tab

This is the dataset that contains the zoning information.

3) Add Parameter

The next step is to set up a published parameter for the user to select their zone type. Right-click on "User Parameters" in the Navigator window and choose Add Parameter.



When prompted, choose a parameter of type Choice (Multiple). Set the name to ZoneCategoryList and the prompt to "Read what zones?" Click on the Configuration button, and select Import. Select the Zone.tab file, and import the ZoneCategory values. Click OK to create the parameter.

For the Default Value, select all the ZoneCategory values.



The optional flag should be unset.

4) Add Tester

Add a *Tester* transformer connected to the Zones Feature Type. The Tester will be used to filter zones and keep only the one chosen by the user.

Open the Parameters dialog for the *Tester*. Set the parameters as follows:

Left Value	User Parameter > ZoneCategoryList
Operator	Contains
Right Value	Attribute > ZoneCategory

Test Clauses		
Left Value	Operator	Right Value
1  \$(ZoneCategoryList)	Contains	 ZoneCategory

5) Add an Aggregator

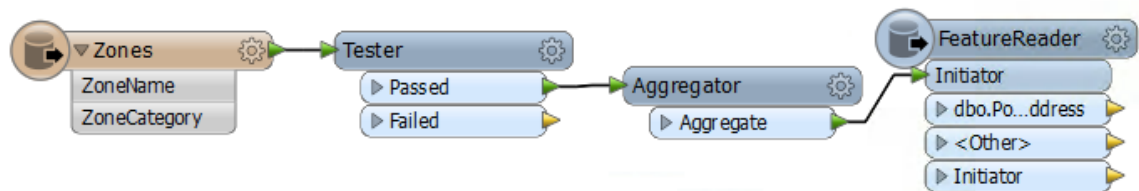
The upcoming *FeatureReader* will issue a request to the database for each feature. The more features you have, the more requests, and this takes time. If you have only one feature, the translation will run faster.

Open the Parameters dialog for the *Aggregator*. Set the following parameter:

Mode: Geometry – Assemble One Level

6) Add FeatureReader

Add a *FeatureReader* transformer connected to the Aggregator. This will be how features are read from the address database table.



7) Set Parameters

Open the parameters wizard for the FeatureReader and set the parameters as follows:

Panel 1:

Reader Format SQL Server Spatial

Click the reader parameters button and enter the database connection parameters as before. Save these settings as the default, if desired.

Click the Table List button and select the PostalAddress table.

Panel 2:

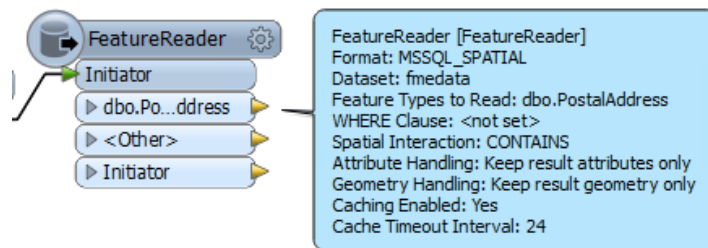
Feature Types Query the Feature Types specified on the previous page
Where Clause <none>

Panel 3:

Spatial Test CONTAINS

Panel 4:

Attribute Handling Result Attributes Only
Geometry Handling Result Geometry Only

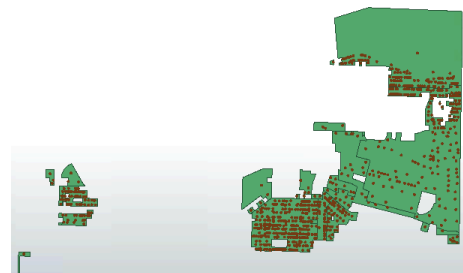


8) Add Inspectors and Run Workspace

Add Inspector transformers to the *FeatureReader* output ports. Save and run the workspace using File > Prompt and Run. When prompted, select Industrial and Light Industrial.

The workspace will read addresses from the SQL Server database, only where they fall inside the chosen census tract.

The output will look like this:



Updating Features



Updating entire tables is simple enough, but updating individual features is a task that requires a little more finesse.

Once information is stored in a database, its source is unlikely to stay static. Changes will occur.

Sometimes an update will involve reloading an entire set of data, totally replacing the existing content. Sometimes the table will also be replaced, and sometimes just emptied (truncated) and refilled. Sometimes individual features will be updated or deleted.

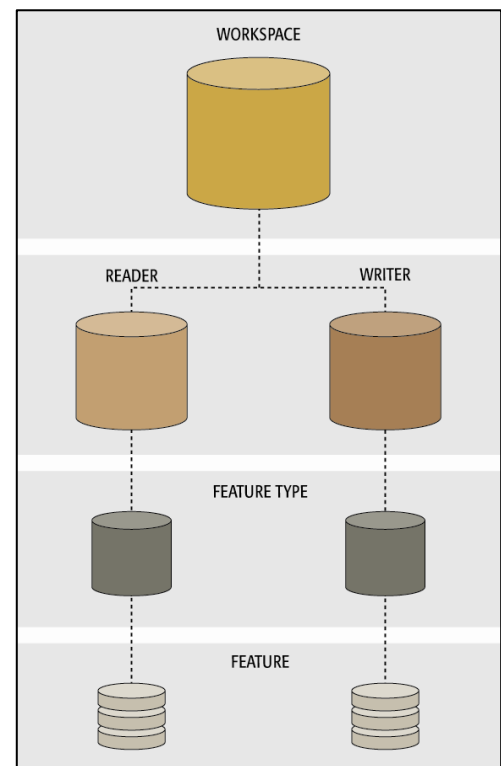
FME provides tools to carry out all sorts of updates and deletions in a database, mostly through parameters.

Controlling Translations

You'll recall that operations on a Reader or Writer (i.e. at the Database level) are carried out using Reader/Writer Parameters, located in the Navigator window.

Operations on a feature type (i.e. at the database table level) are carried out using Feature Type Parameters, located both in the Navigator window and Feature Type Properties dialog (Format Parameters tab).

Operations on individual features in FME are carried out using Format Attributes. Format Attributes are accessed through the Feature Type Properties dialog (Format Attributes tab)



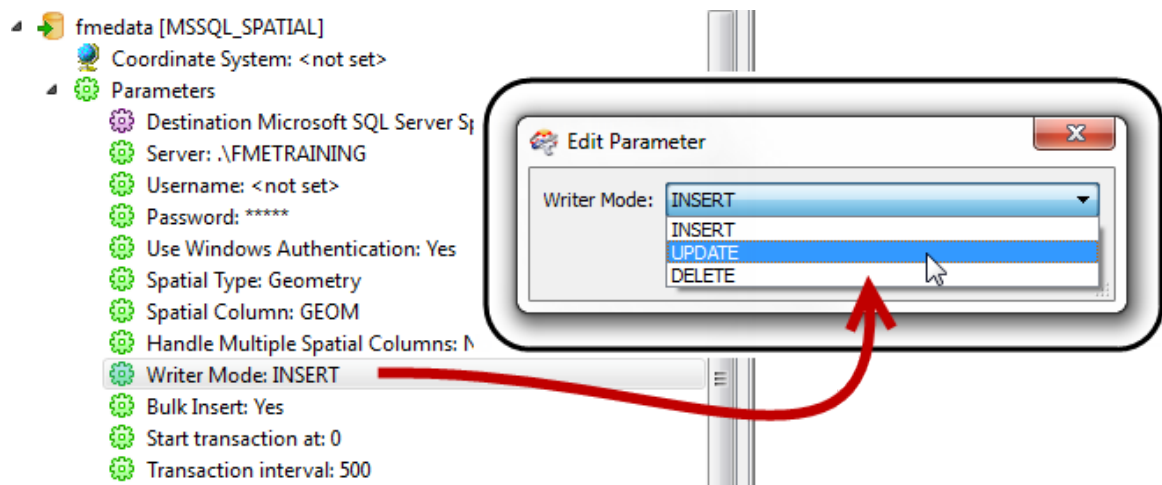
Writer Parameters

Updates and deletions to a database can be primarily controlled through a Writer parameter called Writer Mode. This parameter informs FME what action to carry out on the database. Its three values are INSERT, UPDATE, and DELETE.

INSERT means records are simply added to the database. This can be part of an update where the entire contents of a table are deleted and replaced with new features.

UPDATE means that records are not being inserted or deleted, but simply replaced. Each FME feature written to a database in UPDATE mode replaces an existing database record.

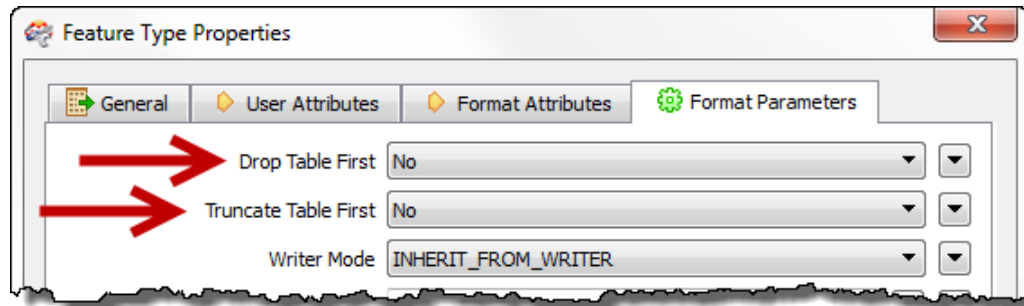
DELETE means that records are being removed (deleted) from a database. Each FME feature written to a database in DELETE mode causes a database record to be deleted.



Feature Type Parameters

Several Feature Type parameters exist to help update existing database tables.

To replace the entire contents of a table, the parameters to use are “Drop Table First” or “Truncate Table First”.

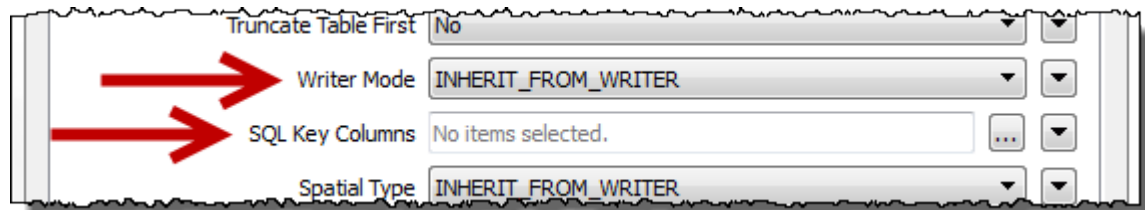


“Truncate Table First” is used when the table needs to be emptied of existing data, but does not otherwise need an update to its schema.

“Drop Table First” is used when the table needs to be emptied AND an update is to be made to the database schema. For example, use this when you wish to update a table with new content and require a new column to be added to the table.

When using either of these, you would want to set the Writer Mode parameter to INSERT. UPDATE and DELETE will be of no use when the existing table has been emptied first.

Two other Feature Type parameters of use are Writer Mode and SQL Key Columns.



Writer Mode acts in the same way as the Writer parameter Writer Mode. The difference is that, as a Feature Type parameter, it only acts on a single table. This is useful for writing to multiple tables using different actions. For example, the mode for one table can be set to INSERT, while the mode for another could be set to UPDATE (and another to DELETE).

The SQL Key Columns parameter is used to select a database column, to specify how incoming features are matched to existing records in an UPDATE or DELETE action.

An UPDATE is carried out when an incoming FME feature has an attribute(s) with the same name and value as the selected column(s), for a record in the database.

Format Attributes

In some cases, UPDATE and DELETE operations will need to be carried out on individual features. Not every record in a table needs to be updated, and not every record will get the same action carried out.

Operations like this – on individual features in FME – are carried out using Format Attributes. For databases there are two particular format attributes that control updates on individual features.

fme_db_operation

fme_db_operation is a format attribute whose value denotes how a database writer should handle that feature. It may take the value DELETE, INSERT or UPDATE.

This format attribute is equivalent to the Writer and Feature Type parameter called Writer Mode. The advantage of using it – instead of those parameters – is that every single feature can be given a different action.

In comparison, the Feature Type parameter forces all features for a particular table to be processed the same way, and the Writer parameter forces all features for all tables to have the same action.

fme_where

fme_where is a format attribute whose value denotes a match that identifies which database record(s) this feature should update.

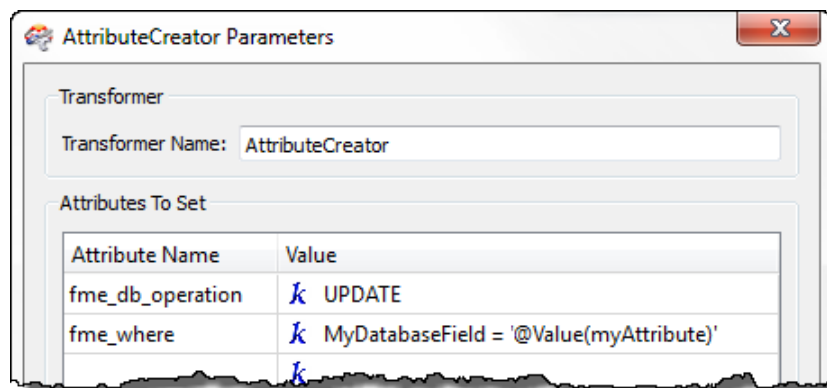
This format attribute is equivalent to the Feature Type parameter called SQL Key Columns. Again, the advantage here is that each individual feature can be given a completely different WHERE clause; whereas the Feature Type parameter applies the same clause to all features.

The structure of this attribute is usually:

```
<database field> <operator> <value>
```

For example a where statement of `MyField = 4` says to update features where the database column named “MyField” has a value of 4.

By creating this string using the AttributeCreator transformer (or the StringConcatenator) the WHERE clause <value> can be obtained directly from an attribute – or attributes – from a published parameter, from an FME Function, or from a more complex string or arithmetic expression.



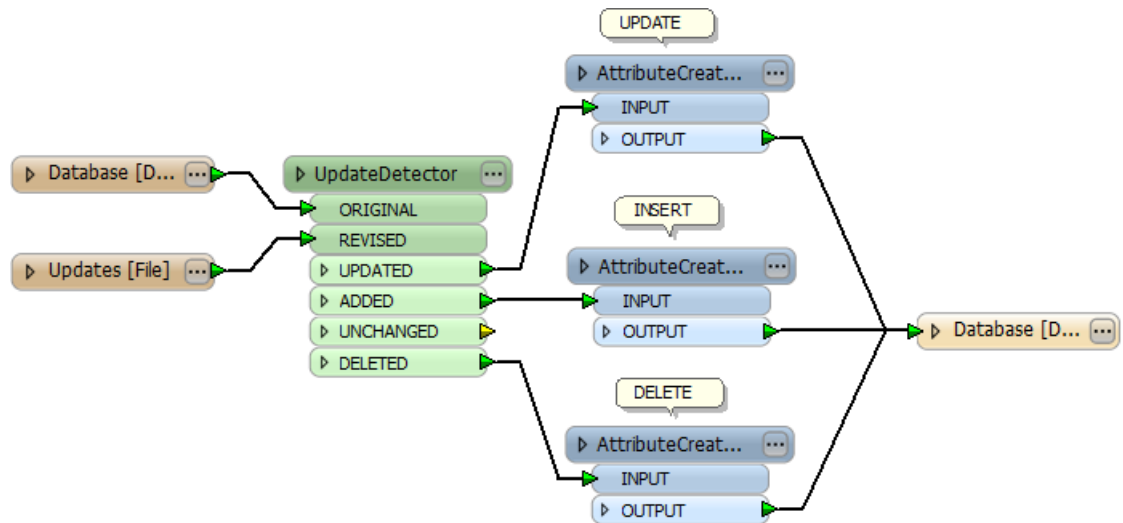
If the attribute is a string, then that part has a quote around it, for example `MyField = 'abc'`

Identifying Features

When an entire database or table needs updating, it's easy to identify which features are to be processed in which way. However, when only certain features need to be updated it's important to be able to identify which features they are.

In this scenario the source data sometimes indicates which features require updates. On other occasions it's necessary to go through a process of change detection.

A typical Change Detection workspace uses a ChangeDetector or Matcher transformer (sometimes in a Custom Transformer like the UpdateDetector) and will look like this:



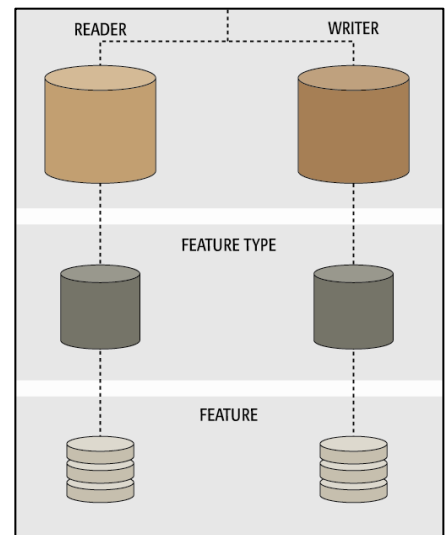
Parameter Priority

The basic rule for parameters is that any higher-level parameter affects every component below it. For example, a Reader Parameter affects all Feature Types that belong to that particular reader.

Database writing mode does work in this way in general. For example, if the writer level is set to INSERT then ALL features are written to tables as an insert.

However, this mode can be set not only at the Writer level, but also at the Feature Type level, or on individual features with a format attribute; and this causes a different effect.

When the same parameter exists at multiple levels, the higher-up parameter only applies when the lower-down parameters are not set (or are set to "INHERIT FROM WRITER"). When the same parameter is set at different levels, then the lower-level parameter wins out.



For example, a Writer might be set as INSERT mode; but a table is set to UPDATE mode. In that case the Feature Type level parameter wins out, and features are written to that table as an update.



Example 5: Feature Updates	
Scenario	FME user; Planning Department
Data	Postal Address Data (ESRI File Geodatabase)
Overall Goal	Load address data and updates
Demonstrates	Feature-level updates
Finished Workspace	C:\FMEData\Workspaces\PathwayManuals\Database2a(MSSQL)-Complete.fmw C:\FMEData\Workspaces\PathwayManuals\Database2b(MSSQL)-Complete.fmw

Planning Department Changes Its Mind

The Retired PostalAddress features should be removed from the database, the Pending should be updated to Current, and the Other should be inserted. The Current and Temporary features do not have to be inserted again.

1) Start FME Workbench

Start Workbench if necessary and open the Exercise 3 workspace. Save it as mssql5.fmw.

2) Assign Operation Type

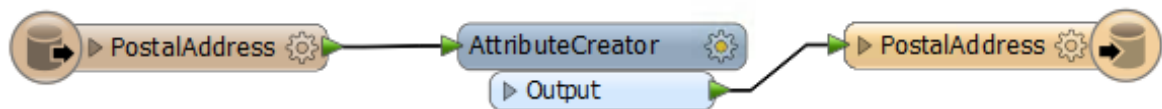
The different action types defined in the updates file need to have different values for *fme_db_operation* in order to carry out the different action for each.

The workspace must be set up to assign the following:

Status	fme_db_operation
Other	INSERT
Retired	DELETE
Pending	UPDATE

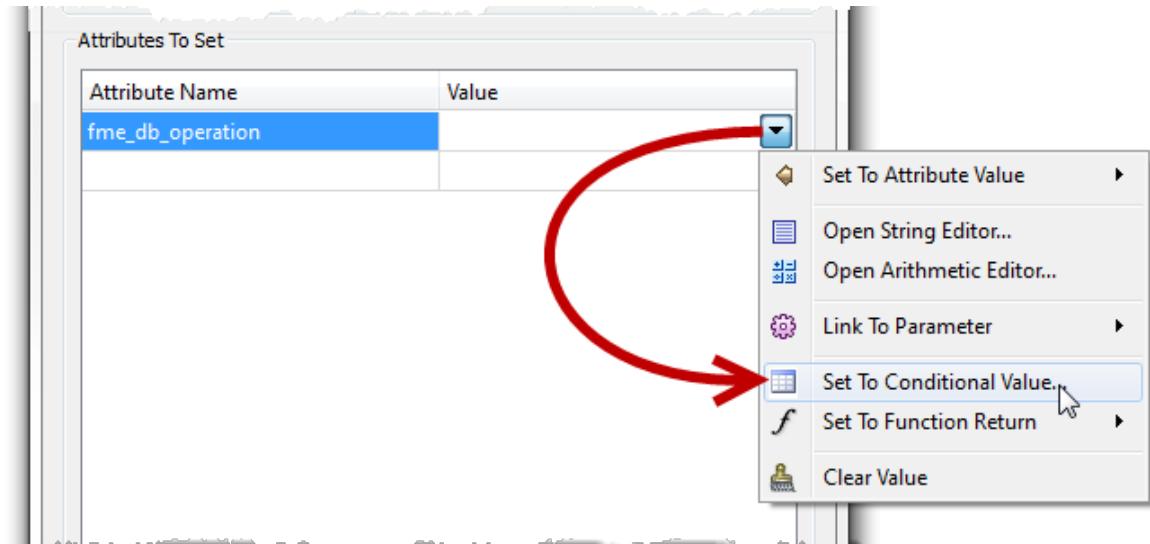
In FME2013-SP1 (or newer) this can be done with an AttributeCreator transformer using new Conditional Mapping functions.

Place an AttributeCreator transformer and connect it to the Reader Feature Type, like so:

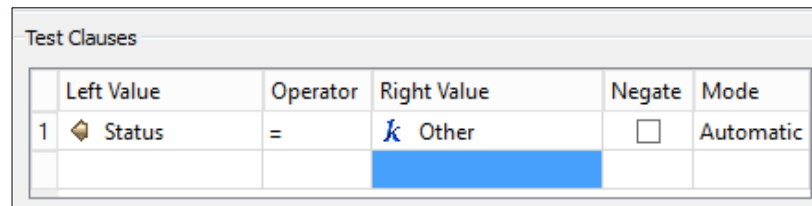
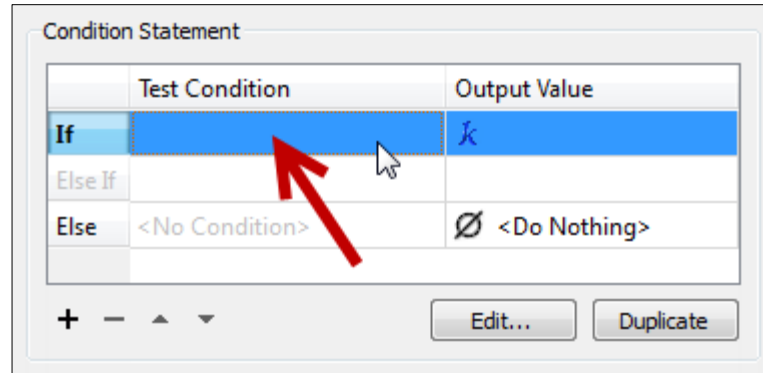


3) Set up INSERT

Open the AttributeCreator parameters dialog. Enter *fme_db_operation* as the Attribute Name to be created. Open the Value drop-down menu and choose "Set to Conditional Value"

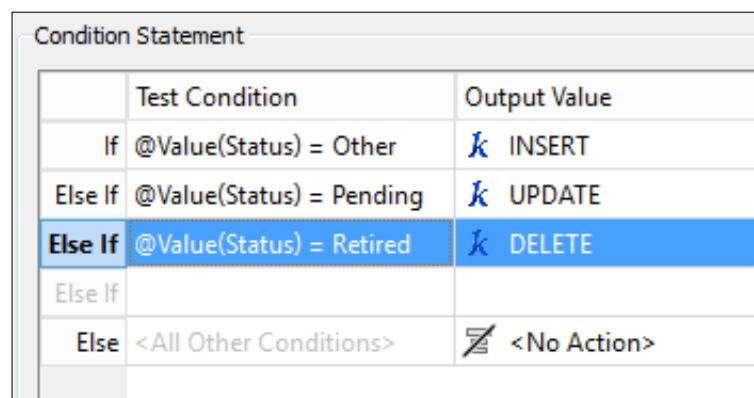
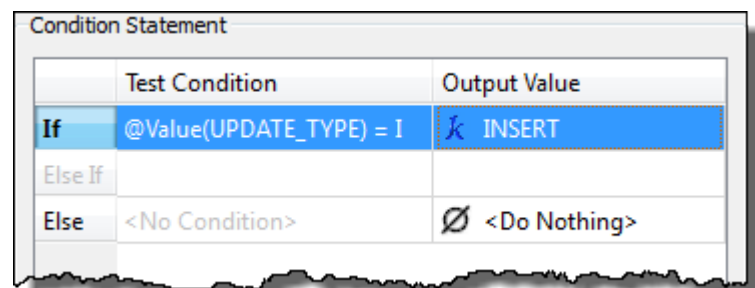


In the Condition Definition dialog there is a line for each test to be carried out. To start with double-click in the first "If" condition.



This opens up a Tester-like dialog. In here enter a test to check whether the incoming feature is an INSERT:

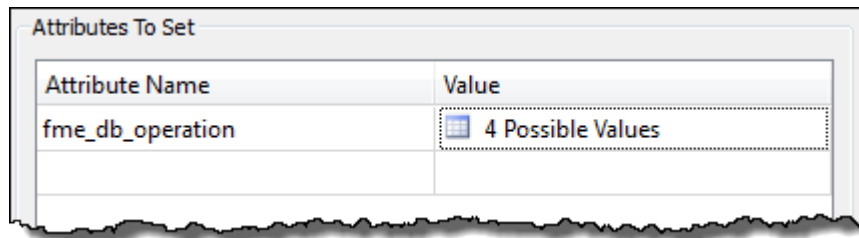
Back in the Condition Definition dialog, set the output value (remember we are setting fme_db_operation here) to INSERT



4) Set up UPDATE/DELETE
Now repeat this process to set up Conditional Mapping for the Pending (UPDATE) and Retired (DELETE) Status types.

5) Assign WHERE Clauses

The AttributeCreator dialog will now show there are 4 possible values for fme_db_operation (INSERT, UPDATE, DELETE, <Do Nothing>).



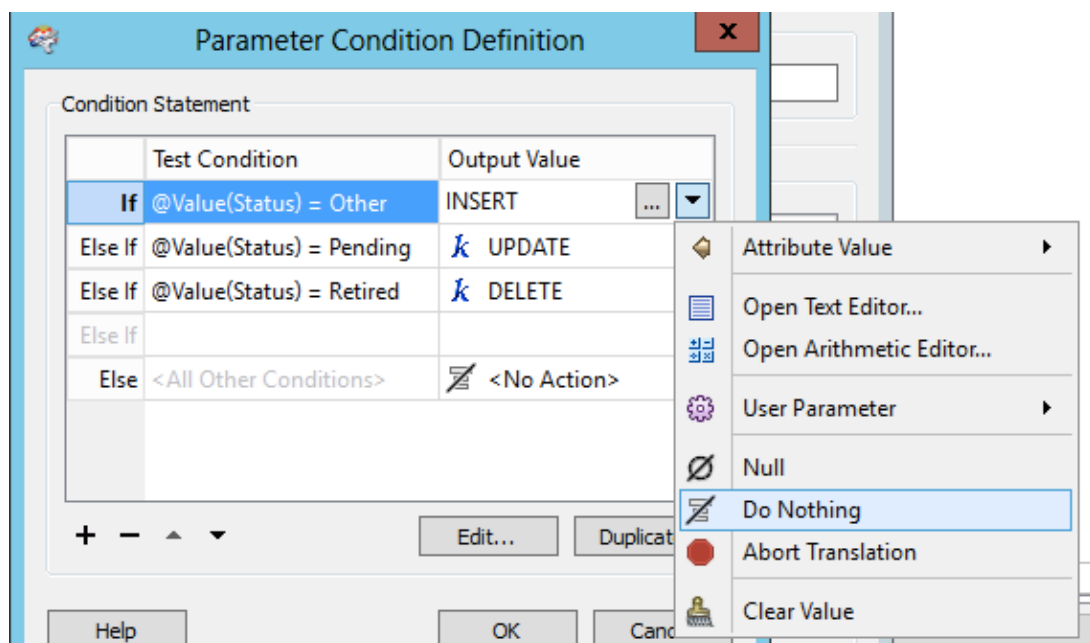
Now we can set up a WHERE clause by creating fme_where.

In the AttributeCreator parameters dialog, click on the entry for fme_db_operation and then click the Duplicate button. This sets up a duplicate set up conditions/values and is the easiest method to use where all of the tests (here for Update Type) will be the same.

Change the newly created attribute name from fme_db_operation(1) to fme_where and select "Set to Conditional Value".

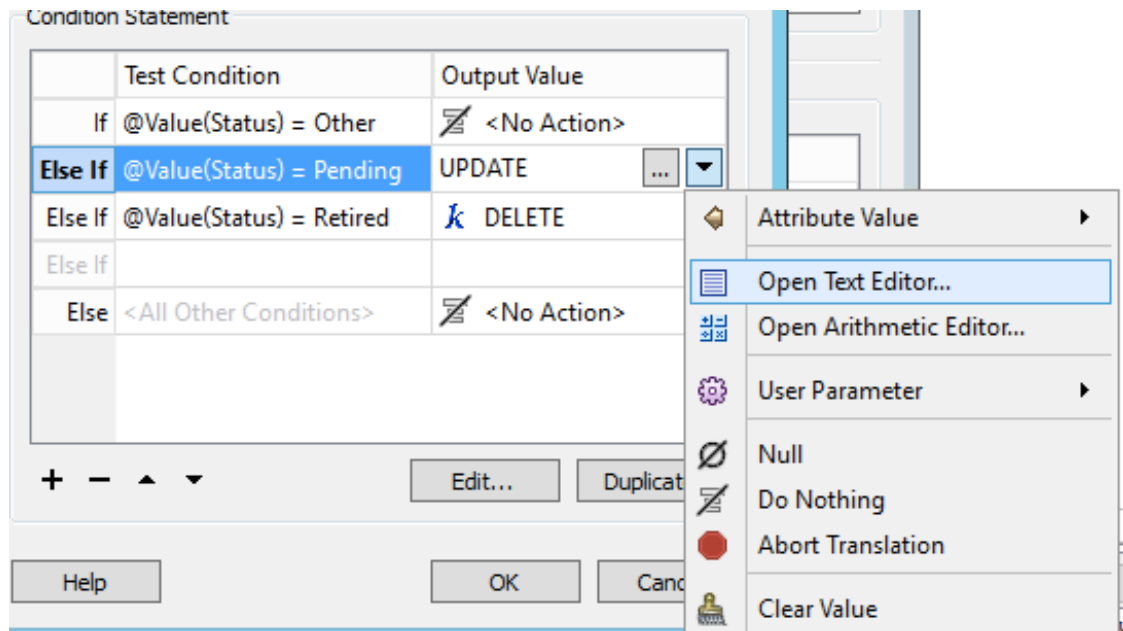
Because this is a set of duplicate conditions, in the Condition Definition dialog we can leave the Test Conditions to be the same and now only need to start editing the output values.

Where Status is "Other", the output value can be left empty, as a WHERE clause is not required for an INSERT. So set the output value to "Do Nothing":





Where Status is "Pending" the output value needs to be a SQL WHERE clause. So click the drop-down menu and select Open String Editor.

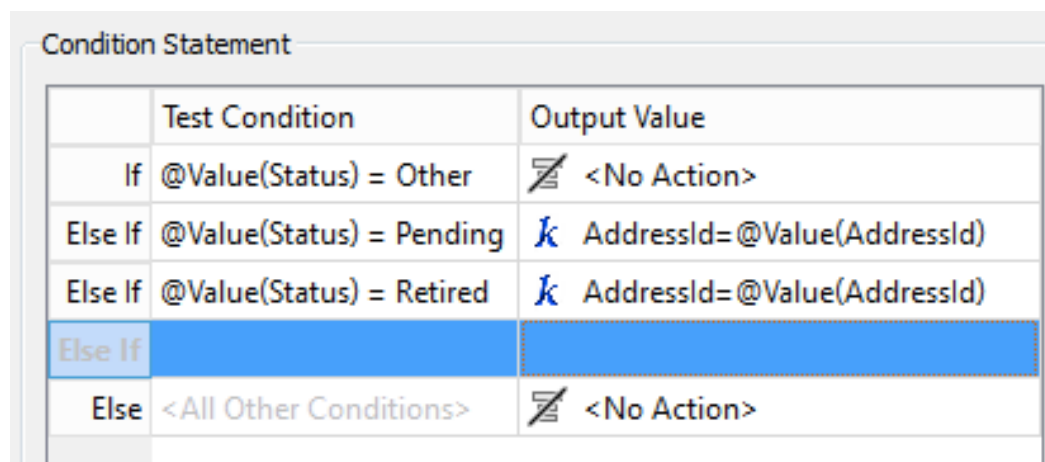


In the String Editor dialog, use either the Basic or Advanced version to create a string matching a field called AddressId to the value of the attribute called AddressId.

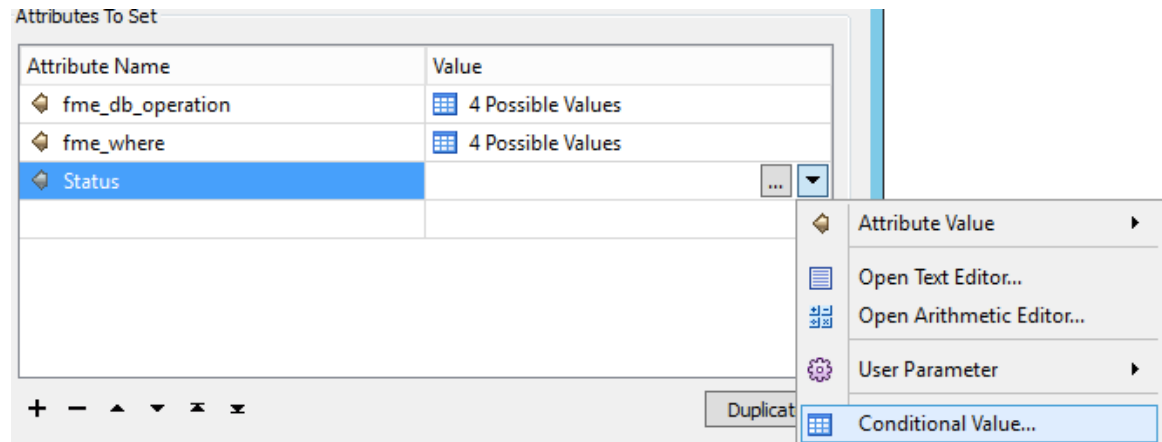
The Advanced Editor will merely show the string: AddressId = @Value(AddressId)

Now repeat the process for the DELETE update type (it will have the exact same WHERE clause):

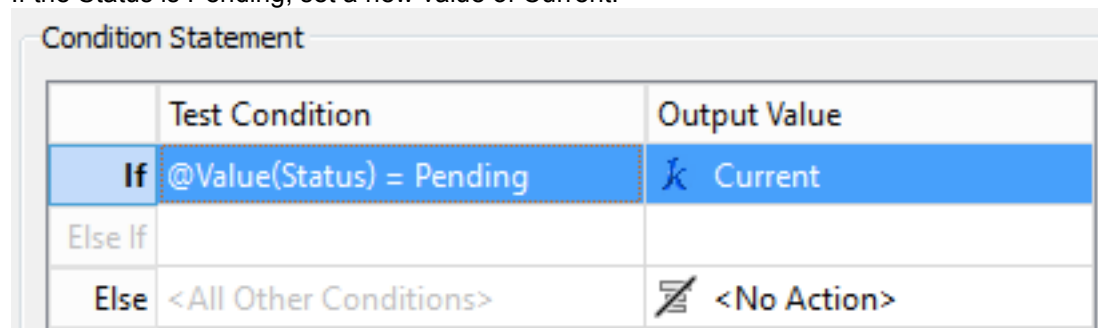
You could, of course, delete the existing DELETE condition and duplicate the UPDATE one.



For the Pending status features, the Status must also be updated to be Current. Another conditional statement will be used to set Status.
Add Status, and set it to a Conditional Value.



If the Status is Pending, set a new value of Current.










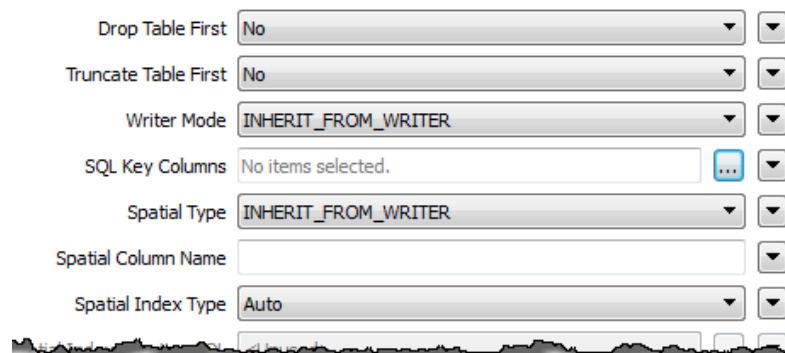
Hit OK to exit the AttributeCreator parameters.

6) Check Parameters

The workspace is now complete, but the parameters need attention.

At this point the Writer (database) parameters probably look like this:

-  Use Windows Authentication: Yes
-  Spatial Type: Geometry
-  Spatial Column: GEOM
-  Handle Multiple Spatial Columns: No
-  Writer Mode: INSERT
-  Bulk Insert: Yes
-  Start transaction at: 0



...and the Feature Type (table) parameters like this:

Make sure that “Drop Table First” and “Truncate Table First” is set to No.

The question now is one of Writer Mode. From the above screenshots you can see there are parameters on both the Writer and the Feature Type. Currently:

Writer: Writer Mode	INSERT
Feature Type: Writer Mode	INHERIT FROM WRITER

We have already defined writer mode (through `fme_db_operation`) to be a separate action (INSERT, UPDATE, DELETE) for individual features. According to the FME Readers and Writers Manual, this will work *“unless the parameter at the feature type level is set to INSERT”!*

The current setup – with feature type mode set (via the writer mode) to INSERT – is not going to give the results we need. We should change the mode to be UPDATE. The two options here are:

- Leave the writer:writer mode parameter as-is, and set the feature type:writer mode parameter to UPDATE.
- Set the writer:writer mode parameter to UPDATE, and leave the feature type:writer mode parameter as-is.




















Choose one of these methods to change writer mode to UPDATE. Are there any obvious benefits to choosing one over the other?

7) Save and Run Workspace

Save the workspace, press “Prompt and Run” and select “Status is not” Current and Temporary, and then run it.

The workspace will give a WARNING because the SQL Server writer is using Bulk Insert mode. This mode allows quicker insertion of data, but it does not allow features to be updated.

Locate the Writer parameter for Bulk Insert and set it to No. This isn’t truly necessary for the translation to work—it will simply remove the warning message.

-  Parameters
 -  Destination Microsoft SQL Server Spatial Name: fmetadata
 -  Server: .\FMETRAINING
 -  Username: <not set>
 -  Password: *****
 -  Use Windows Authentication: Yes
 -  Spatial Type: Geometry
 -  Spatial Column: GEOM
 -  Handle Multiple Spatial Columns: No
 -  Writer Mode: INSERT
 -  Bulk Insert: No 
 -  Start transaction at: 0
 -  Transaction interval: 500
 -  Command Timeout (Seconds): 30
 -  SQL Statement to Execute Before Translation: <not set>
 -  SQL Statement to Execute After Translation: <not set>
 -  Initialize Tables: FIRSTFEATURE
 -  Orient Polygons: Yes

8) Run Workspace (Optional)

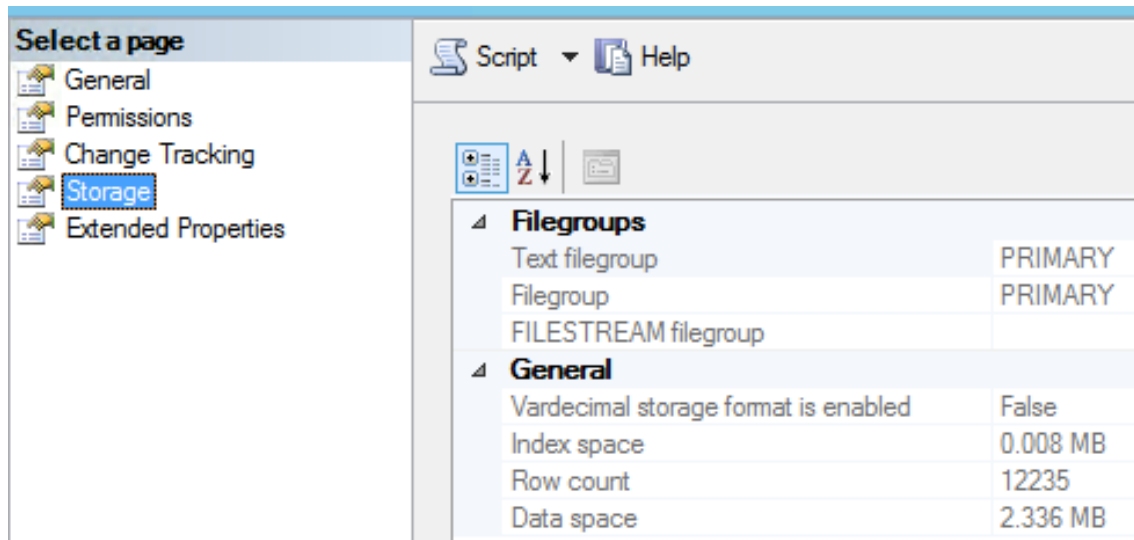
Run the workspace once more.

NB: Each time you run this workspace, it's worth returning first to the original setup workspace, changing the feature type parameter Drop Table First to yes, and then running that. Otherwise we can't be sure that any of the features won't already have been (partly) updated.

Notice that 4,082 features are written to the database. These are composed of:

INSERTS	1387
DELETES	1362
UPDATES	1333

Because ORIGINAL features (12,210) + INSERTS (1387) – DELETES (1362) = 11972, there should now be 12,235 records in the database.



Filegroups	
Text filegroup	PRIMARY
Filegroup	PRIMARY
FILESTREAM filegroup	

General	
Vardecimal storage format is enabled	False
Index space	0.008 MB
Row count	12235
Data space	2.336 MB

Time and Date Attributes in Spatial Databases



Time and Date Attributes are among the more tricky to get into, and out of, a database.

Time and date attributes are complicated territory because each different database format may have its own unique structure for dates.

Microsoft SQL Server

DateTime fields represent date and time data from January 1, 1753 to December 31 9999. For example, a value of 20061231235959 represents 11:59:59PM on December 31, 2006. When writing to the database, the writer expects the date attribute to be in the form YYYYMMDDHHMMSS

SmallDateTime fields represent date and time data from January 1, 1900 to June 6, 2079. For example, a value of 20060101101000 represents 10:10:00AM on January 1, 2006. When writing to the database, the writer expects the date attribute to be in the form YYYYMMDDHHMMSS.

Oracle

fyi: Oracle expects DATE values in the format YYYYMMDDHHMMSS even though when you display a date field from an Oracle table it shows something like this: **01-JAN-08 12:00:00**

Formatting Date Attributes with Transformers

To write dates to a database DATE or DATETIME field you can use the *TimeStamper* or *DateFormatter* transformer to get the date into the correct format.

A format string of `^Y^m^d^H^M^S` will return a date-time in the form YYYYMMDDHHMMSS

A format string of `^Y^m^d` will return a date in the form YYYYMMDD

New for FME2013, the DateFormatter now also allows you to specify the source date format using the same format strings. Also provided are default output strings of the most common output formats.

Coordinate System Granularity



Granularity refers to the level at which different features can be written to different coordinate systems

FME has the unique ability to allow different tables to have different coordinate systems.

The first feature to be written to a table sets the coordinate system for all subsequent features in that table (rather than the entire writer). Therefore each table may have a different coordinate system.

Supported Formats

This functionality is supported in the following database formats:

- Geodatabase and SDE
- SQL Server
- Informix
- Teradata
- IBM DB2

It is not (yet) supported in:

- Oracle
- GeoMedia SQL Server Warehouse
- GeoMedia Access Warehouse

Multiple Geometries



Multiple Geometries are permitted where supported by the database, usually in the form of multiple geometry columns per table

Most databases include the ability to have multiple geometry columns per table, and FME does too. However, the table must exist beforehand – FME cannot create multiple geometry tables.

Multiple Geometry Writing

There are two multiple-geometry writing scenarios:

- Reading AND Writing multiple geometries
- Reading single geometry features and converting them to multiple geometries

In a Multiple -> Multiple translation, the writing is handled automatically.

However, when converting single geometries to multiple, the key is in how to identify two features that are related, and how to assign each of them to the appropriate geometry column.

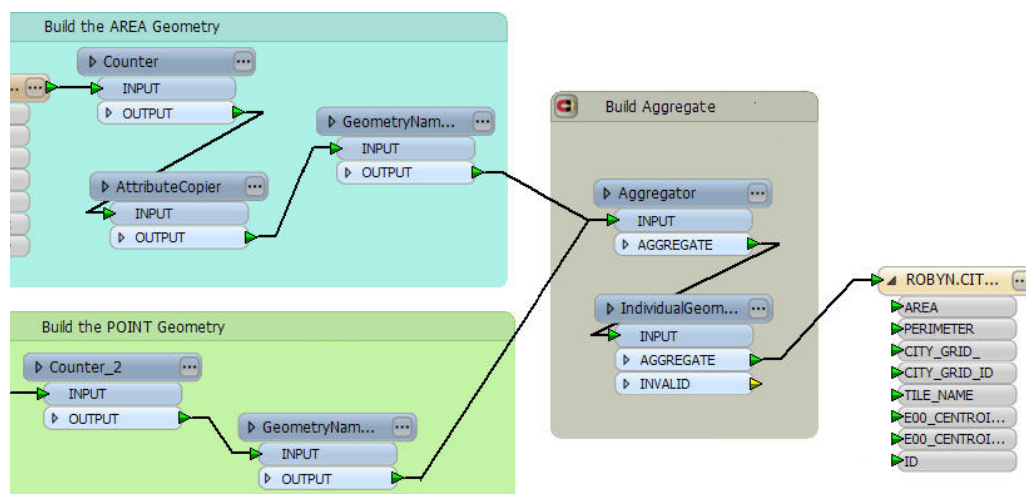
The functionality used to do this involves geometry names and aggregates.

Because FME doesn't (*yet*) support multiple geometries within Workbench, the setup for each database record to be written is a little contrived. It will be composed of two or more features, each of which contributes its geometry to the final record.

A geometry name is applied to each feature (with a *GeometryPropertySetter*) and this identifies the geometry column(s) to write to.

The features are grouped together as an aggregate - usually with an *Aggregator* transformer – and this identifies which features form a particular database record.

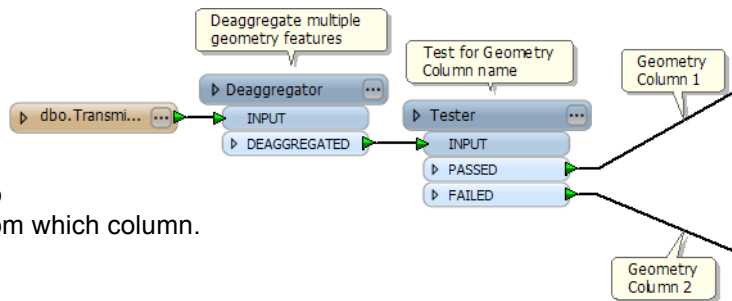
A *MultipleGeometrySetter* transformer is used to tell the writer to treat each feature in the aggregate as a different geometry for the same record.



Multiple Geometry Reading

Similar to writing, multiple geometry reading involves aggregates and lists.

Each multiple geometry feature that FME reads is an aggregate. The *Deaggregator* transformer can be used to split up the record into the individual geometries, and an attribute (`_geometry_name`) used to determine which geometry came from which column.





Example 6: Multiple Geometry Writing	
Scenario	FME user; Planning Department
Data	Parks (MapInfo TAB, SQL Server)
Overall Goal	Write to multiple geometry columns
Demonstrates	Multiple Geometry Writing
Finished Workspace	C:\FMEData2014\Workspaces\Database\mssql6-complete.fmw
SQL Script	C:\FMEData2014\Workspaces\Database\TableCreate.sql

In this example, FME will be used to create multiple geometries where the two geometries are different representations (point and polygon) of the same park objects.

A workspace will then be created to read the data back, and filter it (either points or polygons) depending on the scale required for the output.

1) Start FME Workbench

Start FME Workbench, and open the Generate Workspace dialog.

Set up a translation as follows:

Reader Format MapInfo TAB (MITAB)
Reader Dataset C:\FMEData2014\Data\Parks\Parks.tab

Writer Format Microsoft SQL Server Spatial

Click the writer parameters button. In the parameters dialog enter the database connection parameters. Click **OK**, and then **OK** again, to create the workspace.

2) Create Table

FME isn't (yet) able to create tables with multiple geometry columns, so this must be done with a piece of SQL code. However, FME can run such code, and that is how we will create the table.

Locate and double-click the writer parameter 'SQL Statement to Execute Before Translation'. Enter the following script:

```
FME_SQL_DELIMITER ;
-DROP TABLE Parks;
CREATE table Parks (
    "ParkId"           INTEGER,
    "RefParkId"       INTEGER,
    "ParkName"        CHAR (40),
    "NeighborhoodName" CHAR (40),
    "EWStreet"        CHAR (30),
    "NSStreet"        CHAR (30),
    "DogPark"         CHAR (1),
    "Washrooms"       CHAR (1),
    "SpecialFeatures" CHAR (1),
    "POINTS"          GEOGRAPHY,
    "POLYGONS"        GEOMETRY,
);
```

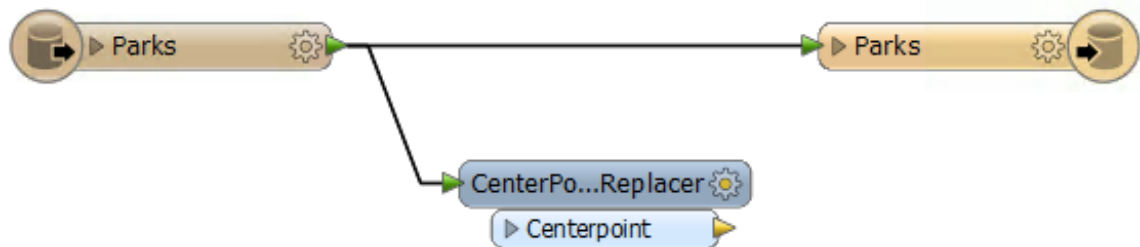
The script will check if the table exists and, if so, drop it before continuing. Then it will recreate the table with multiple geometry fields; one for points, one for polygons.

3) Add CenterPointReplacer

Now to create the multiple geometries: The park features are originally polygons; to create points insert a *CenterPointReplacer* transformer, connected to a second output stream from the source feature type.

The transformer has no parameters (except name) to worry about.

The workspace will now look something like this:



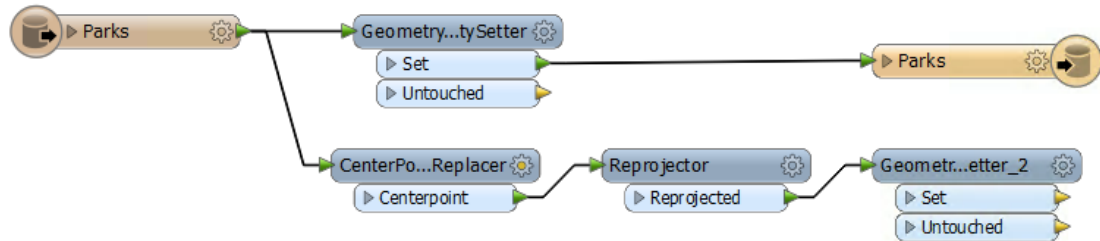
4) Add Reprojector

The point geometry will be stored as a geography spatial type, which must be in a round-earth coordinate system. Insert a Reprojector transformer after the CenterPointReplacer, and set the Destination Coordinate System as LL84.

5) Add GeometryPropertySetter

Now there are two sets of geometries, and they must be given different names. The names should match the geometry columns names in the database: POINTS and POLYGONS

Place two *GeometryPropertySetter* transformers as shown:



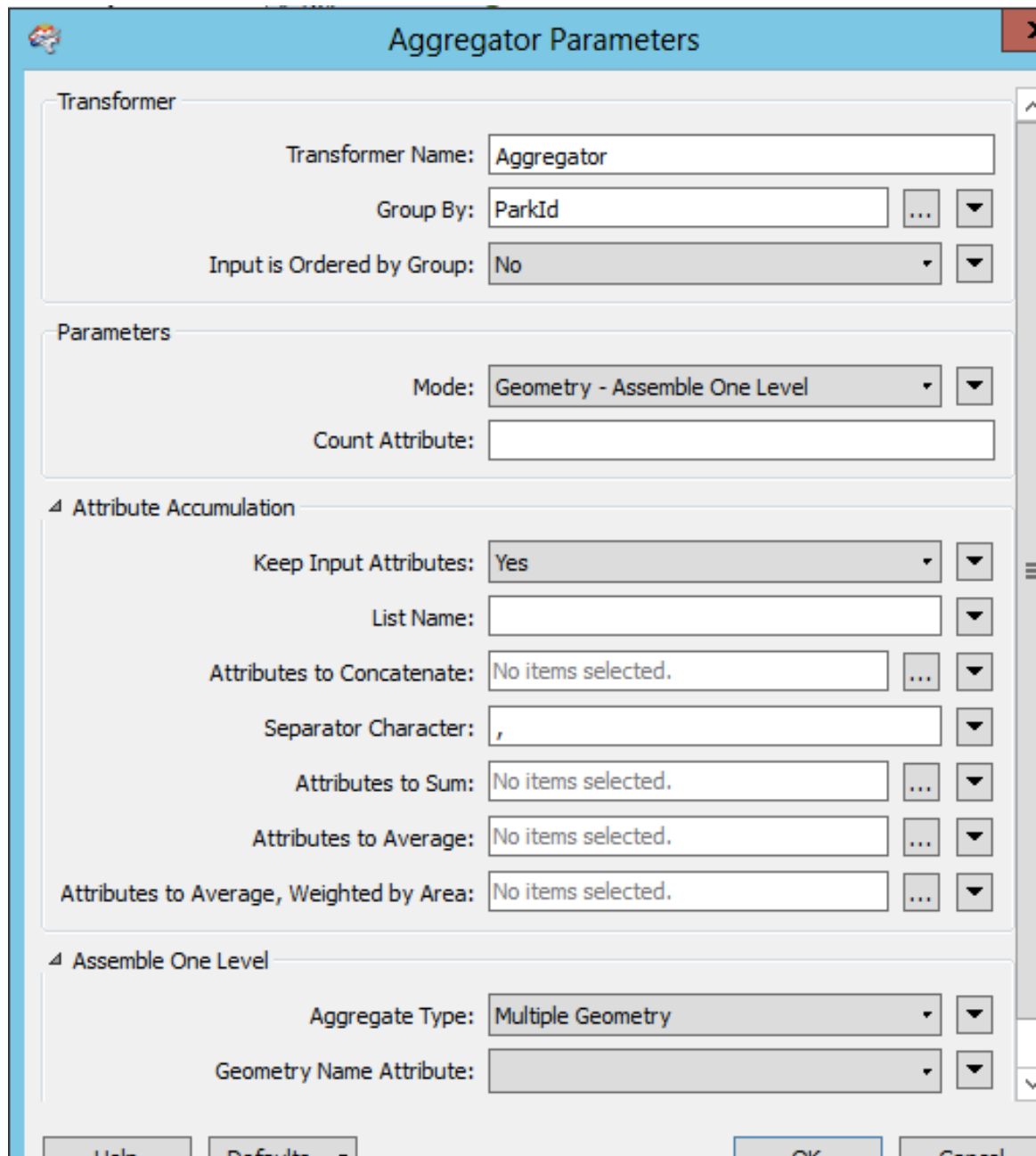
Open the Parameters dialog for each transformer in turn.
Change the Property to Set parameter to Geometry Name.

Enter the required geometry name in the field provided (either POLYGONS or POINTS)

7) Add Aggregator

The next step is to identify the matching features. This is done with an *Aggregator* transformer. Place an *Aggregator* transformer and connect both streams of data to it.







In the parameters, set *group_by* to use *ParkId*; this is how the two sets of features are paired off and aggregated together. Set the Mode as 'Geometry-Assemble One Level', and set Keep Input Attributes to Yes.



Set the Aggregate Type parameter to “Multiple Geometry” so that FME knows these are multiple geometry features,

8) Set Multiple Geometry Parameter

One last task: In the Navigator window, locate the Writer parameter *Handle Multiple Spatial Columns*, and set it to Yes.

-  Spatial Type: Geometry
-  Spatial Column: GEOM
-  Handle Multiple Spatial Columns: Yes 
-  Writer Mode: INSERT
-  Bulk Insert: Yes

9) Save and Run Workspace

Save the workspace and then run it. The workspace will check for a table of that name, delete it if necessary, create it anew, and then fill the table with multiple-geometry features.

10) Start SQL Server Management Studio

Start the SQL Server Management Studio and log in.

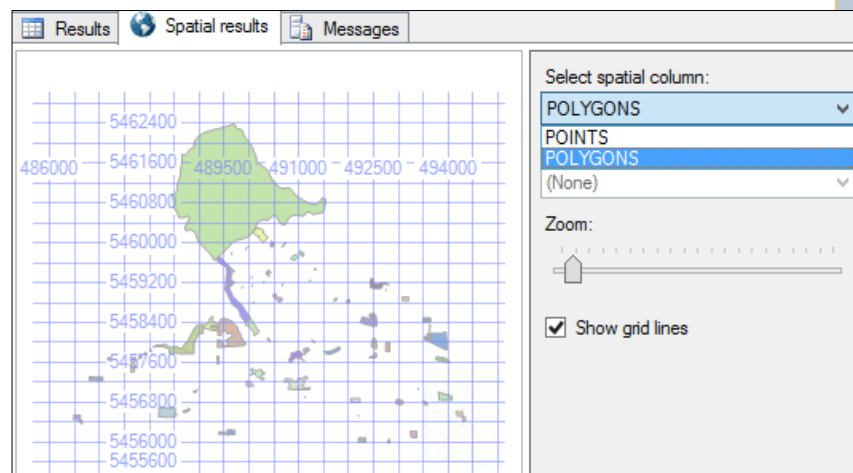
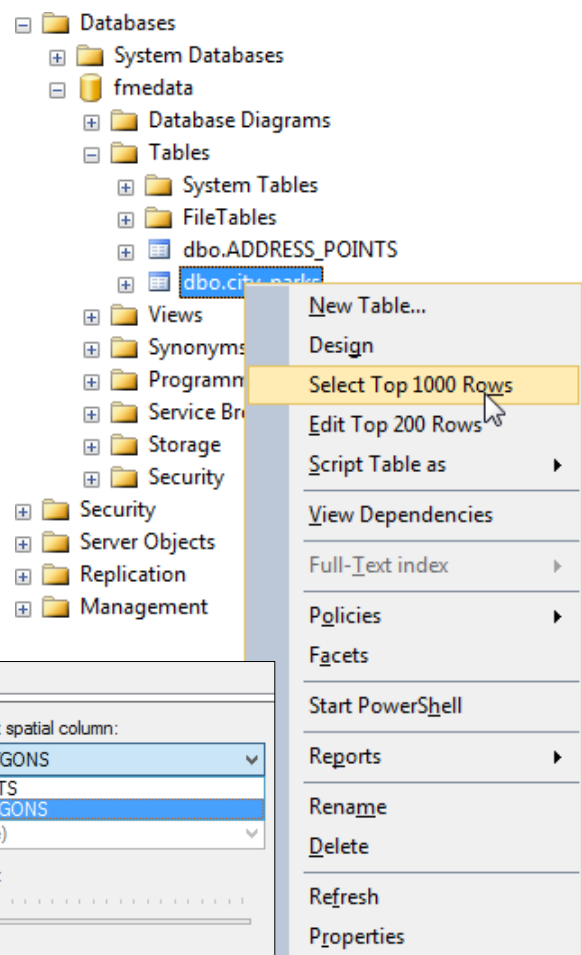
11) Browse for Database

In the Object Explorer window, browse to the fmedata database, and then browse to the Parks table.

Right-click on the table name, and choose the option to *Select Top 1000 Rows*.

Notice that each record has two geometry columns:

Click the Spatial Results tab, and flip between the two spatial columns:





Example 7: Multiple Geometry Reading	
Scenario	FME user, Planning Department
Data	Parks (SQL Server, KML)
Overall Goal	Read and filter multiple geometry columns
Demonstrates	Multiple Geometry Reading
Finished Workspace	C:\FMEData2014\Workspaces\Database\mssql7-complete.fmw

In this example, FME will be used to read the previous data back, and filter it (either points or polygons) depending on the scale required for the output.

1) Start FME Workbench

Start FME Workbench, and open the Generate Workspace dialog. Set up a translation as follows:

Reader Format Microsoft SQL Server Spatial

In the Reader parameters dialog enter the database connection parameters, then select Parks as the table to read.

Writer Format Google Earth KML
Writer Dataset C:\FMEData2014\Output\Training\Parks.kml

Click **OK** to create the workspace.

2) Set Multiple Geometry Parameter

In the Navigator window, locate the reader parameter *Handle Multiple Spatial Columns*, and set it to Yes.

3) Add Deagggregator

Add a Deagggregator transformer. This will divide the data into its two geometry types. Notice that one of the parameters is for *Geometry Name Attribute*.

4) Add Tester

Add a *Tester* transformer. Set up a test to check if the geometry name attribute (by default *_geometry_name*) has a value of POINTS.

To prove that the workspace is doing what is expected (so far) attach an *Inspector* transformer to each *Tester* output port and run the translation. Points and polygons should get separated out.

5) Add Reprojector

The features from the SQL Server database do not contain coordinate system information. You will have to add this to the features. Add a *CoordinateSystemSetter* after each *Tester* output. The points are in LL84, and the polygons are in UTM83-10.

Advanced Task

Although it's no longer database related, let's work on making the KML display the different features at different zoom levels.

6) Add KMLRegionSetters

Add two KMLRegionSetter transformers; one for the points, one for the polygons.

The point's version should be set to:

Bounding Box: Calculate:	Yes – 2D
Buffer XY Region By	0.003
Minimum Display Size	10
Maximum Display Size	30

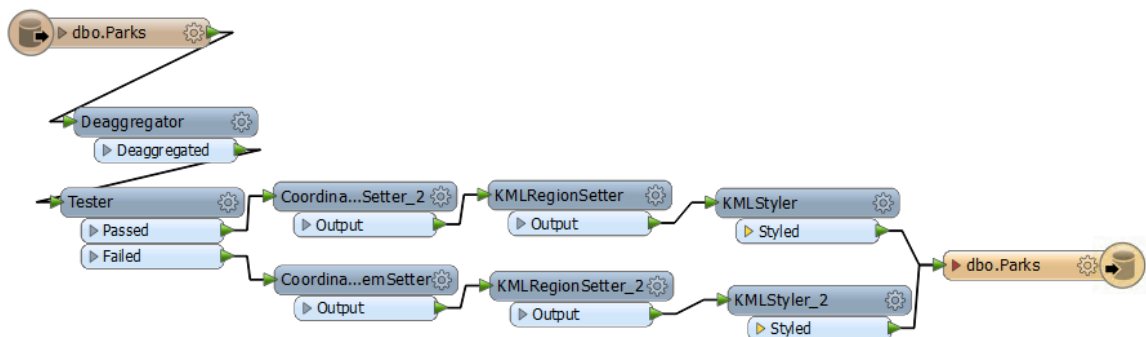
The polygon's version should be set to:

Bounding Box: Calculate:	Yes – 2D
Minimum Display Size	50
Maximum Display Size	-1

7) Add KMLStylers

Add two KMLStyler transformers; one for the buffered points, one for the polygons. Assign the buffered points a circular red icon (e.g. H1), and the polygons green.

The workspace will now look like this:



8) Save and Run Workspace

Save the workspace and then run it. Open the output in Google Earth.

The 'point' geometry will show when the view is zoomed out; as it is zoomed in the points will disappear to be replaced by the actual polygons. You can experiment with the different display sizes and the point buffer parameter if you wish to fine-tune the result.

Database Transformers



Besides the FeatureReader there are a number of database-related transformers for submitting SQL statements directly to the database.

There are two methods to submit SQL statements to a database. Firstly, SQL statements can be entered into parameters to run before and after a translation. Secondly, there are SQL-related transformers.

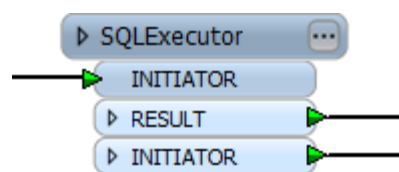
Such statements might be used to:

- Create, drop, modify or truncate a database table
- Carrying out a database join
- Drop constraints prior to data loading
- Any other function that is usually carried out using a SQL statement.

This section will focus on the use of transformers to run SQL statements.

SQLExecutor

The SQLExecutor is a transformer for executing SQL statements against a database. Each incoming INITIATOR feature triggers the SQL statement that has been defined.



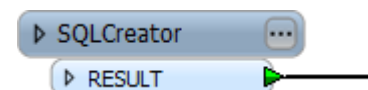
If the SQL is a query, and if features are returned from the database, those features form the output from the transformer. There will be a feature output for each row of the results.

The transformer also exposes result attributes, and does not need to be followed by an *AttributeExposer*.

SQLCreator

The SQLCreator transformer is similar to the SQLExecutor, but does not rely on incoming features to trigger it. Instead, the statement is executed once only.

Like the SQLExecutor, there will be a feature output for each row of the results.





Example 8: SQLExecutor	
Scenario	FME user; Planning Department
Data	Address Data (SQL Server) Fire Halls (GML)
Overall Goal	Carry out a join using the SQLExecutor
Demonstrates	SQLExecutor transformer
Finished Workspace	C:\FMEData2014\Workspaces\Database\mssql8-complete.fmw

The city has a dataset of Fire Halls in GML format.

In this example, FME will be used to update the address database, to flag addresses that are a Fire Hall. This will be done by using a SQLExecutor transformer to do a database join.

1) Inspect Source Data

Use the FME Data Inspector to open and inspect the source file:

Reader Format GML (Geography Markup Language)
Reader Dataset C:\FMEData2014\Data\Emergency\FireHalls.gml

Use the Table View window to view the text records. Notice that each Fire Hall facility has an address field. We'll try to match this to the SQL Server address table.

	gml_parent_id	gml_id	HallNumber	Name	Address	PhoneNumber
5	<missing>	ida1d84f2f-d4f0...	6	Vancouver Fire ...	1001 Nicola St	604-665-6006
6	<missing>	idcc686e5b-c5a...	7	Vancouver Fire ...	1090 Haro St	604-665-6007
7	<missing>	id8bf85022-9d5...	8	Vancouver Fire ...	895 Hamilton St	604-665-6008
8	<missing>	idf6c8bb01-8db...	12	Vancouver Fire ...	2460 Balaclava St	604-665-6012

2) Start FME Workbench

Start Workbench if necessary and begin with an empty workspace.

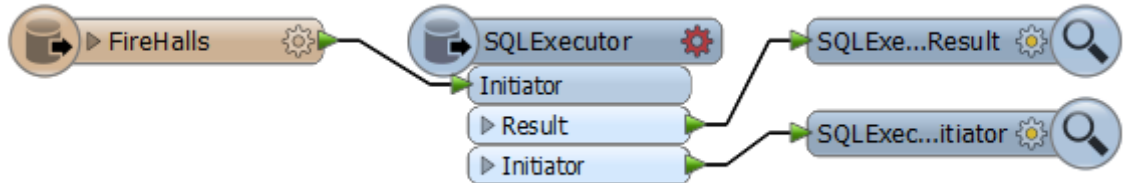
3) Add Reader

Add a Reader using Readers > Add Reader. Set it up as follows:

Reader Format GML (Geography Markup Language)
Reader Dataset C:\FMEData2014\Data\Emergency\FireHalls.gml

4) Add SQLExecutor

Connect an *SQLExecutor* transformer.



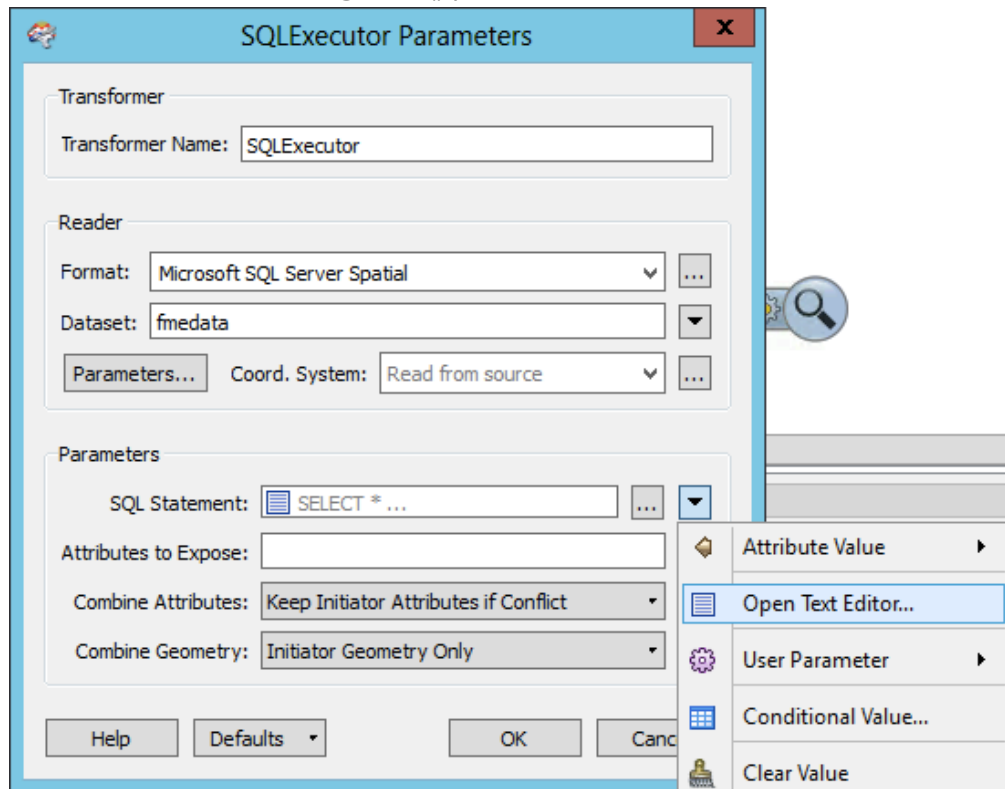
Open the *SQLExecutor* parameters dialog. Set up the parameters as follows:

Reader Format Microsoft SQL Server Spatial
Reader Dataset fmedata

SQL Statement `select * from dbo.PostalAddress where PostalAddress='@Value(Address)'`

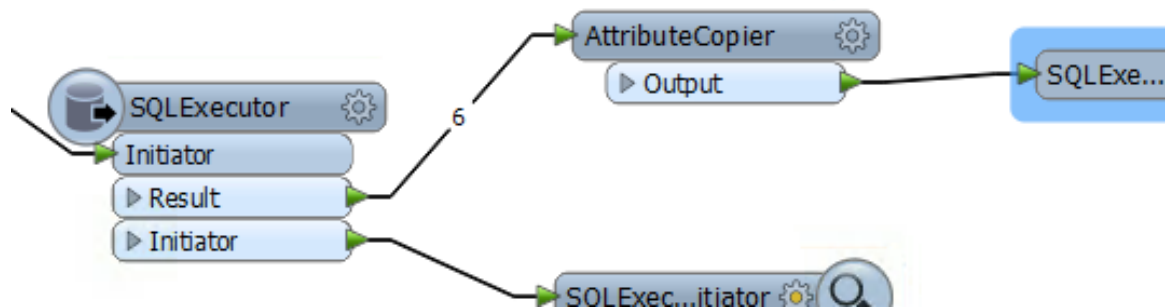
Combine Attributes Keep Initiator Attributes if Conflict

The SQL statement is most easily created by using the editor tool. Be sure to include the quote characters around the final @Value() part!



7) Add AttributeCopier

Add an AttributeCopier transformer to copy *Name* to *OwnerName1*



OwnerName1 is a field in the SQL Server Address table.

If we were to write the data this would cause FME to populate the name of the Fire Hall OwnerName1 field of the PostalAddress table.

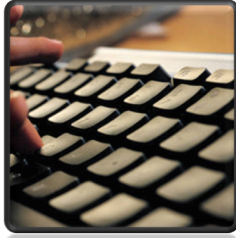
8) Save and Run Workspace

Save the workspace and then run it. Inspect the output to confirm the *OwnerName1* field now includes the contents of the Fire Hall Name attribute.

Advanced Task

If you have the time, use what you know about issuing updates to a database table to update the SQL Server PostalAddresses that are Fire Hall facilities. You'll have to use a FeatureHolder transformer, otherwise the table will be locked by SQLExecutor while the writer is trying to write.

Geometry



Like any spatial format, it's important to be aware of what geometries are supported within SQL Server and how to clean bad geometries to prevent translation failures.

Supported Geometries

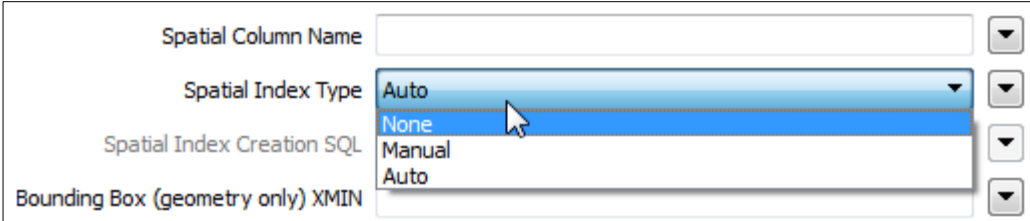
Like any other format, the list of supported geometries is listed in the FME Readers and Writers Manual. For SQL Server the following are listed as supported:

- Points
- Lines
- Polygons and Donut Polygons
- Aggregates
- Circular Arcs
- Curves
- Globes (read only)

Z values and Measures are supported as of FME2013.

Spatial Indices

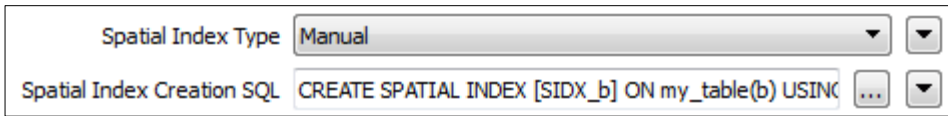
Spatial Indices can be created on a SQL Server table with FME by using a Feature Type Parameter called "Spatial Index Type".



The three options are None, Auto, and Manual.

Automatic creation of a spatial index – possible only with SQL Server 2012 or later – uses a set of default options and a set of Bounding Box coordinates to be specified in the same dialog.





However, due to the complexity of Spatial Index Creation, FME allows a user to specify a SQL statement to create an index on a specific spatial column with non-default options:



Examples of SQL spatial index creation can be found in the FME Readers and Writers Manual.

Geometry or Geography?

SQL Server supports both Geometry and Geography spatial types. The type of geometry can be specified either at the Writer level (see Navigator Window):

-  Use Windows Authentication: Yes
-  Spatial Type: Geometry 
-  Spatial Column: GEOM

...or at the Feature Type level:

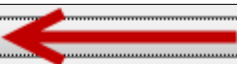
Spatial Type	GEOMETRY	▼
Spatial Column Name	INHERIT_FROM_WRITER	▼
Spatial Index Type	Manual	▼

Notice that there is also the option for the table to inherit the spatial type from the Writer level, in a similar way to the Writer Mode parameter.

Cleaning Geometry

SQL Server is known to be very exacting about the structure of geometry being loaded. If the data does not conform to SQL Server standards then an error is likely to occur.

The GeometryValidator transformer can be used to check for and repair features that have invalid geometry structures. There are various modes of operation on this transformer; the one to start with in this case should be Basic Geometry Integrity, which will clean up features with inherently corrupt or badly formed geometry (rather than quirky structures simply not tolerated by SQL Server). Self-intersections can also cause problems when writing to the GEOGRAPHY model.

Set of Issues to Detect: **Basic Geometry Integrity** 

Issue	Parameters	Repairable
<input checked="" type="checkbox"/> Contains NaNs or Infinities		Yes
<input checked="" type="checkbox"/> Contains Null Geometry Parts		Yes
<input type="checkbox"/> Duplicate Consecutive Points	...	Yes
<input checked="" type="checkbox"/> Degenerate or Corrupt Geometries	...	Yes
<input type="checkbox"/> Self-Intersections in 2D		Yes
<input type="checkbox"/> Non-Planar Surfaces	...	Yes
<input type="checkbox"/> Invalid Solid Boundaries		Yes
<input type="checkbox"/> Invalid Solid Voids		Yes
<input type="checkbox"/> Fails OGC Simple		No
<input type="checkbox"/> Fails OGC Valid		No
<input type="checkbox"/> Missing Texture Coordinates		Yes
<input type="checkbox"/> Missing Vertex Normals		Yes

Performance



Performance is a key concern for most database users, and manipulating how records are inserted and committed is one way in which performance can be improved.

Default Performance

The default behavior of the SQL Server writer is to send one feature at a time to the database.

Once all features have been loaded in this way, the data is committed; meaning that the data is made permanent in the database.

Transaction Interval

Transaction Interval is a SQL Server writer parameter that specifies the number of individual features to be written to the database before a commit action takes place.

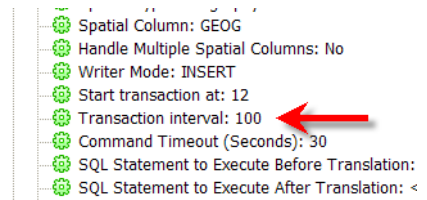
The default Transaction Interval, zero, causes all features to be loaded before being committed. A value of one causes each individual feature to be committed by itself.

If writing to a database from FME fails for some reason, then the data loading process is rolled-back (undone) to the previous commit point; so setting a transaction interval to commit data at regular intervals ensures the entire process is not rolled-back because of one bad feature.

However, increasing the interval can help performance, because it takes fewer database transactions to actually load data. Therefore setting this parameter becomes a trade-off between performance and insurance.



Any data written by the SQLExecutor is NOT considered to be part of the same transaction as that written by a writer.



Bulk Insert

A further SQL Server writer parameter is Bulk Insert. By default, Bulk Insert is enabled when you add a new writer.

Bulk Insert changes the behavior of FME to write features as a batch, rather than individually. This can improve performance because it reduces the network traffic between FME and database to just one set of communications, rather than a communication for every single feature. This is particularly significant when loading data to Azure, the internet/cloud-based version of SQL Server. In tests, Bulk Insert mode produced performance improvements of 400x-1000x.

For SQL Server, Bulk Insert is tied to Transaction Interval. The number of features written as a batch is equal to the value of the Transaction Interval parameter; so <n> features are loaded into the database at a time and then committed at once.

Bulk Insert works with both the Spatial and Non-Spatial versions of the SQL Server writer.



As a previous example showed, the only writer mode compatible with Bulk Insert is INSERT. Bulk Insert is not possible when the writer mode is set to either UPDATE or DELETE.

Other Bulk Insert dependencies are:

- .NET Framework v4
- SQL Native Client 2008 or greater
- Microsoft System CLR Types 2012

Start Transaction At

Whenever a translation fails mid-write, the current transaction is rolled-back. FME will report in the log window how many transactions had been applied up to that point, and what the transaction to restart at should be.

Then, once the problem is fixed, the translation can be re-run, but specifying which translation to start at. The process is then easier and quicker because the database doesn't need to recommit any data except that in the failed transaction.



```

Spatial Column: GEOG
Handle Multiple Spatial Columns: No
Writer Mode: INSERT
Start transaction at: 12
Transaction interval: 100
Command Timeout (Seconds): 30
SQL Statement to Execute Before Translation:
SQL Statement to Execute After Translation: <
    
```

If the "Transaction to Start Writing At" parameter is set to zero – the default – then all data is written as usual.

Some writers also support a format attribute called `fme_db_transaction` that can be used to control commits and rollbacks at the feature level. See the Readers and Writers manual for more documentation on this functionality.

Session Review



This session was all about spatial databases and FME.

What You Should Have Learned from this Session

The following are key points to be learned from this session:

Theory

- Connecting to a database requires a set of **connection parameters** that may vary from database to database
- Updating features is done with two FME **Format Attributes**
- **Reader Parameters** can be used to improve data reading performance
- FME can read and write **multiple geometries**, but not create a table with multiple geometry columns
- **Transactions** help deal with performance and failover

FME Skills

- The ability to connect to a database, write data, and update individual features
- The ability to use reader parameters, both alone and with concatenated parameters
- The ability to read and write multiple geometries
- The ability to use the SQLExecutor and SQLCreator transformers