FME Desktop ®

Database (SQL Server) Pathway Training

FME 2013-SP1 Edition

SAFE SOFTWARE

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 – 2013 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

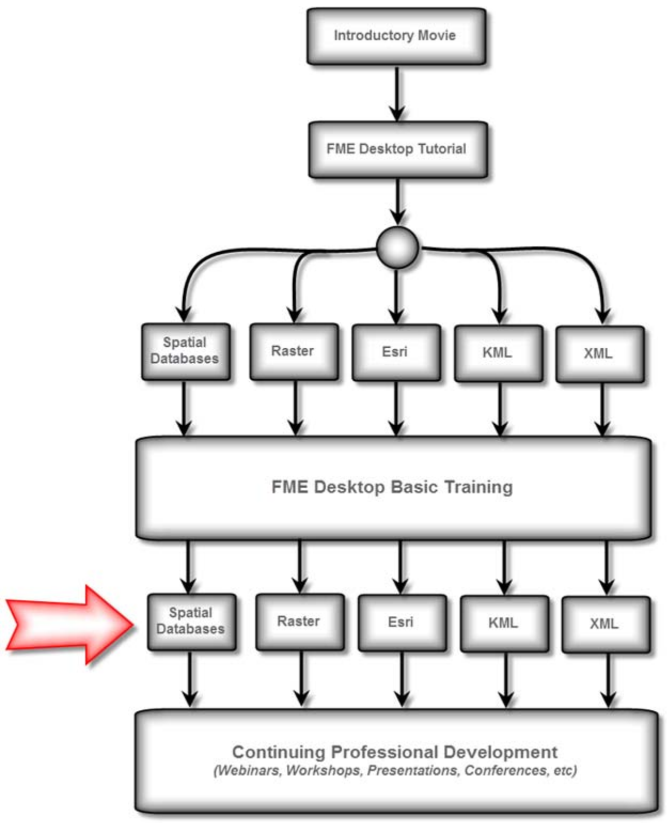
Document Name: FME Desktop Database Pathway Training Manual
FME Version: FME 2013-SP1 (Build 13448) 32-bit
Operating System: Windows 7 SP-1, 64-bit
Database: Microsoft SQL Server Express 2012, 64-bit
Updated: April 2013

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.


Introductory Movie
FME Desktop Tutorial
Spatial Databases
Raster
Esri
KML
XML
FME Desktop Basic Training
Spatial Databases
Raster
Esri
KML
XML
Continuing Professional Development
(Webinars, Workshops, Presentations, Conferences, etc)

FME Version
This training material is designed specifically for use with FME2013-SP1. 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

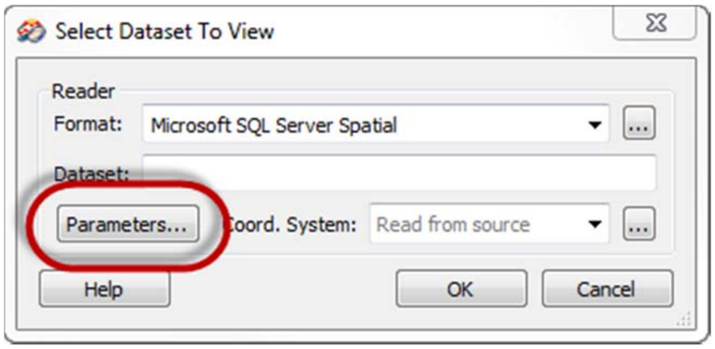
Connecting to the database is the one step all FME operations must perform.

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.

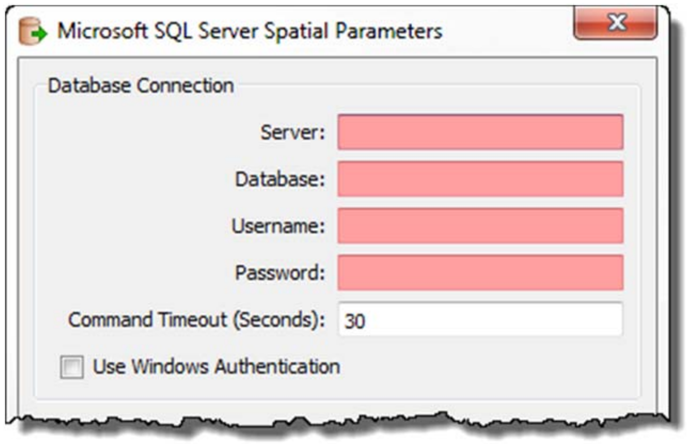
Basic Connection Parameters
The basic connection parameters are:

- 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.



Select Dataset To View
Reader
Format: Microsoft SQL Server Spatial
Dataset:
Parameters…
Coord. System: Read from source
Help
OK
Cancel



Microsoft SQL Server Spatial Parameters
Database Connection
Server:
Database:
Username:
Password:
Command Timeout (Seconds): 30
Use Windows Authentication

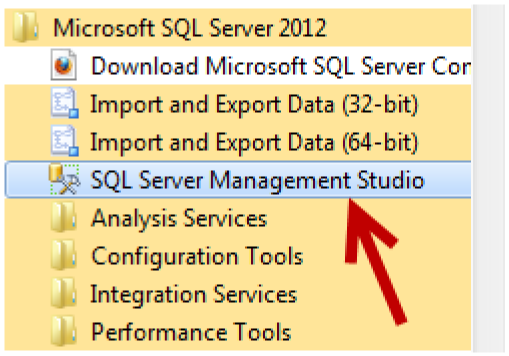
Connecting to SQL Server
Connecting to a SQL Server database requires just four of the basic connection parameters. Port number is not required.

There is an additional option to use Windows Authentication.

Example 1: Connection Parameters	
Scenario	FME user; City of Interopolis, Planning Department
Data	City 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:\FMEData\Workspaces\PathwayManuals\Database1(MSSQL)-Complete.fmw

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

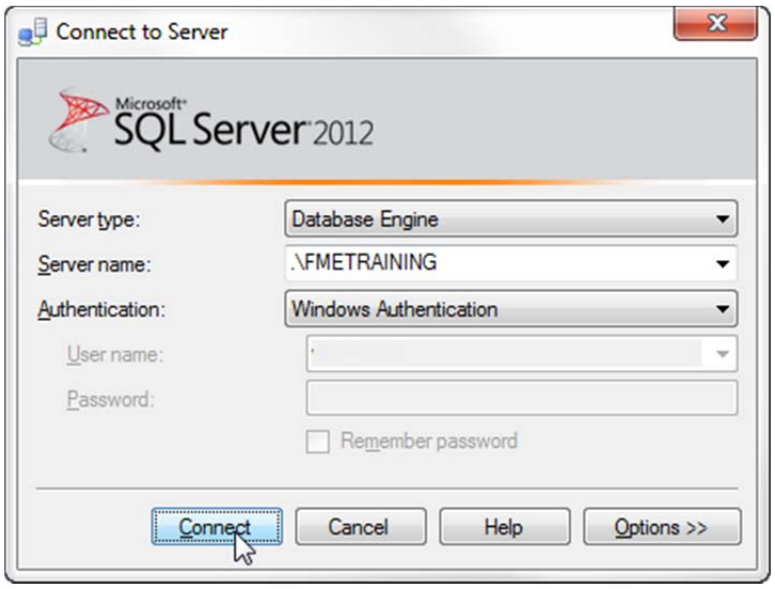
1) Start SQL Server Management Studio
Start the SQL Server Management Studio.
It is found under *Start > All Programs > Microsoft SQL Server 2012 > SQL Server Management Studio*


Microsoft SQL Server 2012
Download Microsoft SQL Server Cor
Import and Export Data (32-bit)
Import and Export Data (64-bit)
SQL Server Management Studio
Analysis Services
Configuration Tools
Integration Services
Performance Tools

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 a server on the local machine, you may use a single dot character, like so:

.\FMETRAINING



Server type: Database Engine
Server name: .\FMETRAINING
Authentication: Windows Authentication
User name:
Password:
Remember password

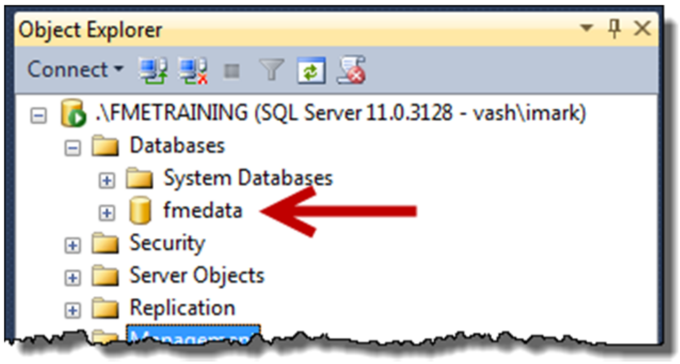
If that fails, simply use a dot (period) character by itself.

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 (MFAL)
Reader Dataset	*C:\FMEData\Data\Parks\city_parks.tab*
Writer Format	Microsoft SQL Server Spatial

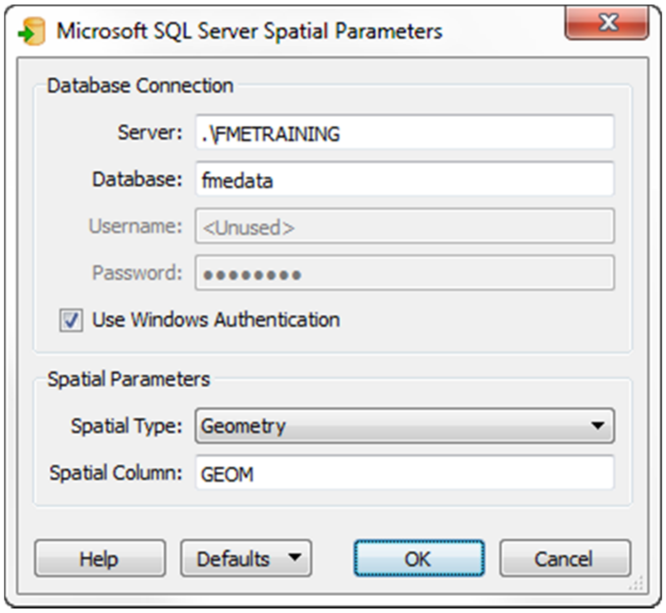
Click the writer parameters button. In the parameters dialog enter the database connection parameters.

In FME training this will usually be:

Server	.\FMETRAINING
Database	fmedata
Use Windows Authentication	Yes

***NB:** Be sure to use a backslash character, and not a forward-slash.*

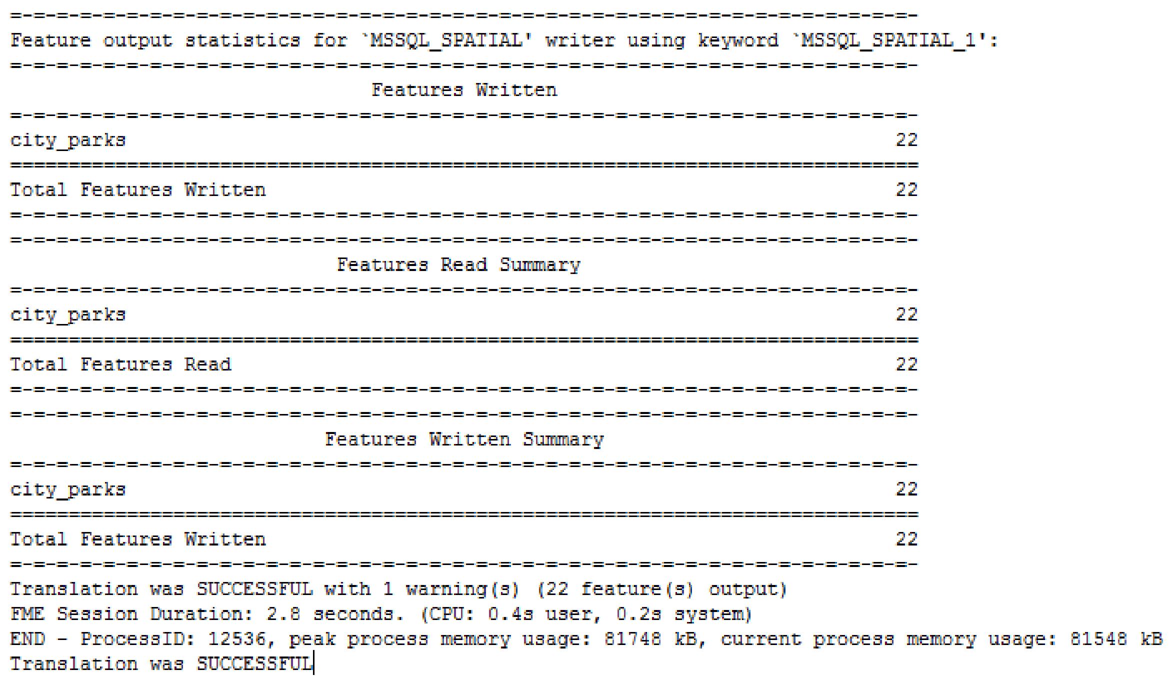
Click **OK**, and then **OK** again, to create the workspace.


Microsoft SQL Server Spatial Parameters

Database Connection
Server: .\FMETRAINING
Database: fmedata
Username: <Unused>
Password: ••••••••
☑ Use Windows Authentication

Spatial Parameters
Spatial Type: Geometry
Spatial Column: GEOM

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



```
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
Feature output statistics for `MSSQL_SPATIAL' writer using keyword `MSSQL_SPATIAL_1':
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
                               Features Written
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
city_parks                                                                   22
==============================================================================
Total Features Written                                                       22
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
                            Features Read Summary
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
city_parks                                                                   22
==============================================================================
Total Features Read                                                          22
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
                           Features Written Summary
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
city_parks                                                                   22
==============================================================================
Total Features Written                                                       22
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
Translation was SUCCESSFUL with 1 warning(s) (22 feature(s) output)
FME Session Duration: 2.8 seconds. (CPU: 0.4s user, 0.2s system)
END - ProcessID: 12536, peak process memory usage: 81748 kB, current process memory usage: 81548 kB
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.

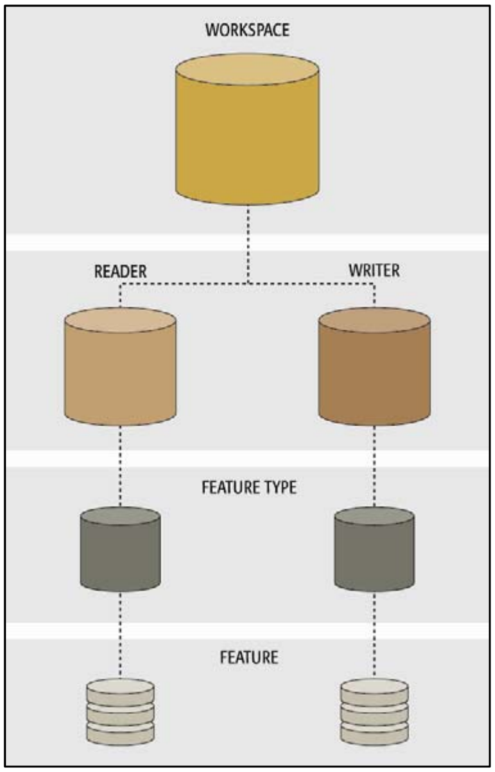
Updating Features

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 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)

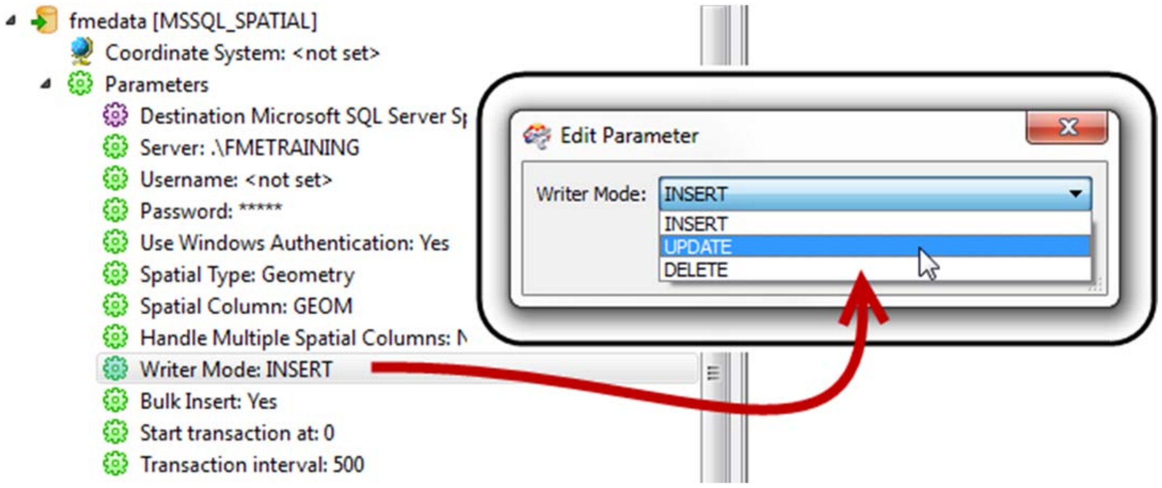
Writer Parameters

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.

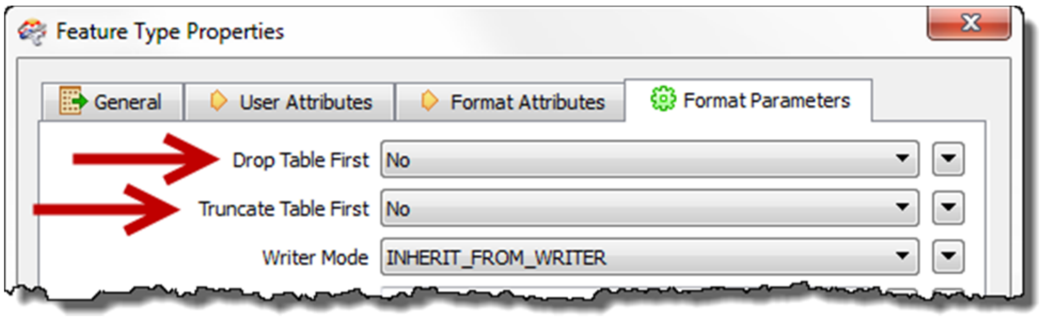
DELETE means that records are being removed (deleted) from a database. Each FME feature written to a database in DELETE mode causes a database record to be deleted.



fmedata [MSSQL_SPATIAL]
Coordinate System: <not set>
Parameters
Destination Microsoft SQL Server Sp
Server: .\FMETRAINING
Username: <not set>
Password: *****
Use Windows Authentication: Yes
Spatial Type: Geometry
Spatial Column: GEOM
Handle Multiple Spatial Columns: N
Writer Mode: INSERT
Bulk Insert: Yes
Start transaction at: 0
Transaction interval: 500

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” and “Truncate Table First”.



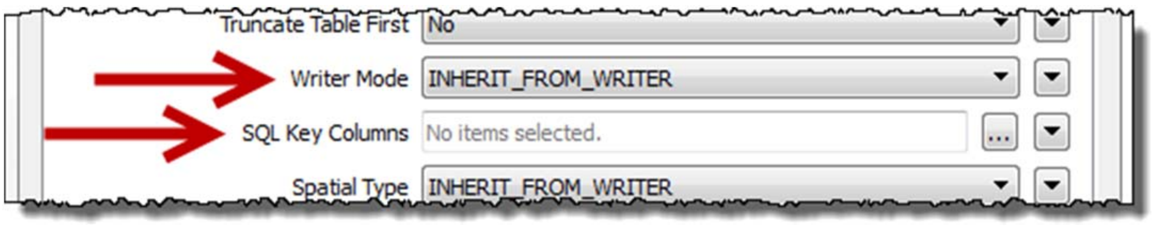
Drop Table First: No
Truncate Table First: No
Writer Mode: INHERIT_FROM_WRITER

“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.



Truncate Table First: No
Writer Mode: INHERIT_FROM_WRITER
SQL Key Columns: No items selected.
Spatial Type: INHERIT_FROM_WRITER

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.

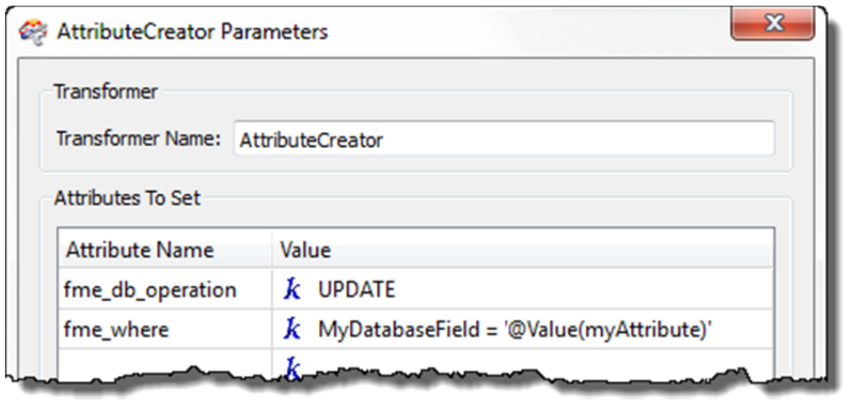
This format attribute is equivalent to the Feature Type parameter called SQL Key Columns. Again, the advantage here is that each individual feature can be given a completely different WHERE clause; whereas the Feature Type parameter applies the same clause to all features.

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.


AttributeCreator Parameters
Transformer
Transformer Name: AttributeCreator
Attributes To Set

Attribute Name	Value
fme_db_operation	UPDATE
fme_where	MyDatabaseField = '@Value(myAttribute)'

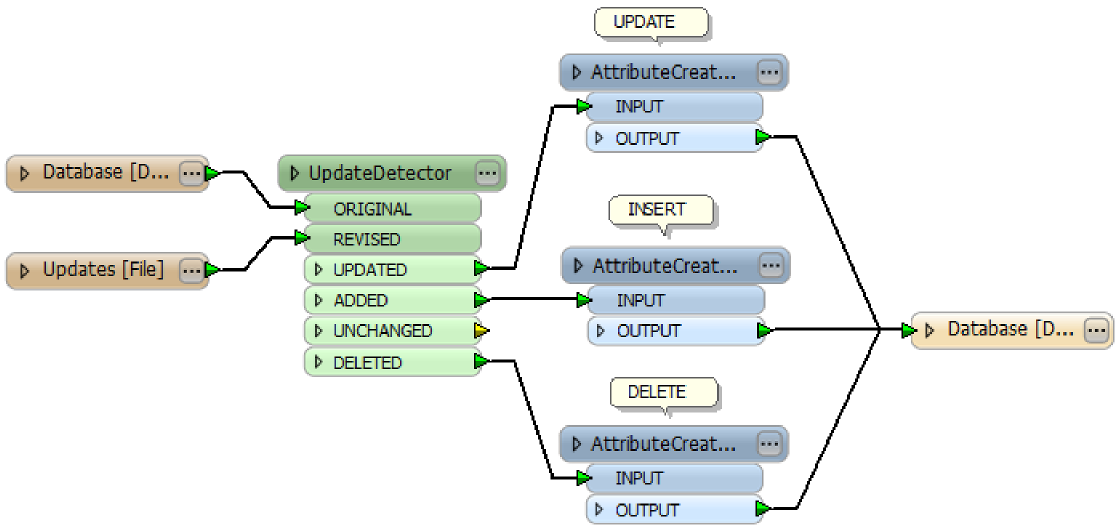
If the attribute is a string, then that part has a quote around it, for example MyField = ‘abc’

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:



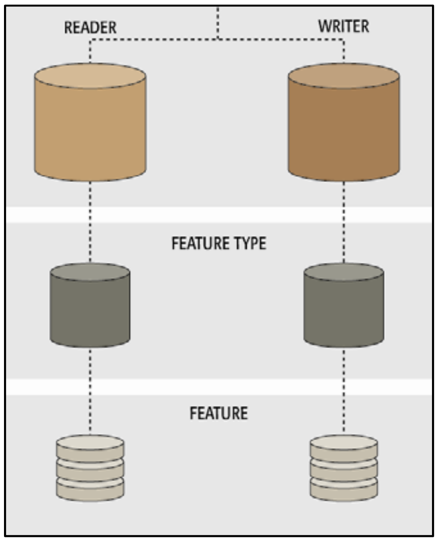
Parameter Priority

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.



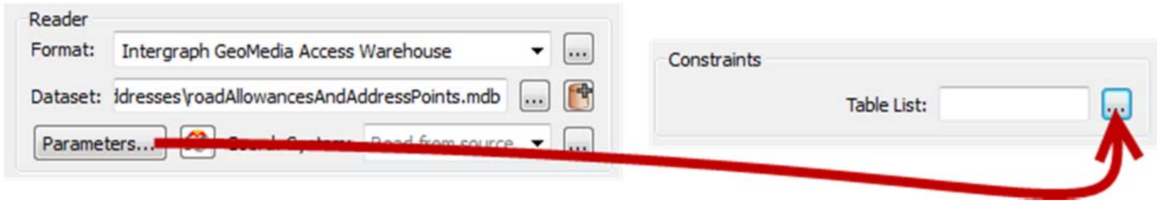
For example, a Writer might be set as INSERT mode; but a table is set to UPDATE mode. In that case the Feature Type level parameter wins out, and features are written to that table as an update.

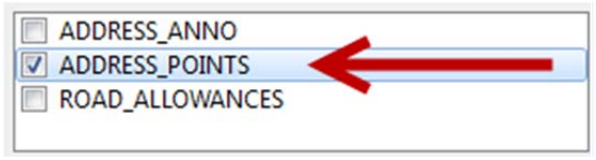
Example 2: Feature Updates	
Scenario	FME user; City of Interopolis, Planning Department
Data	Address Data (GeoMedia Access Warehouse)
Overall Goal	Load address data and updates
Demonstrates	Feature-level updates
Finished Workspace	C:\FMEData\Workspaces\PathwayManuals\Database2a(MSSQL)-Complete.fmw C:\FMEData\Workspaces\PathwayManuals\Database2b(MSSQL)-Complete.fmw

The city has a set of address data, plus a set of updates. Follow these steps to load the data and then apply the updates:

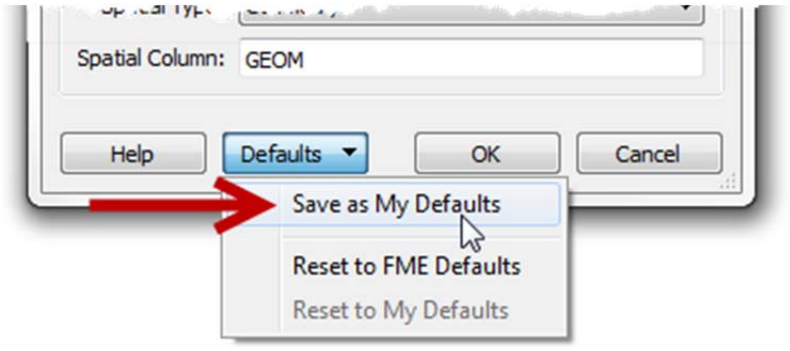
1) Start FME Workbench
Start Workbench if necessary and open the Generate Workspace dialog.
Set up a translation as follows:

Reader Format	Intergraph GeoMedia Access Warehouse
Reader Dataset	*C:\FMEData\Data\Addresses\roadAllowancesAndAddressPoints.mdb*
Reader Parameter	Table List: ADDRESS_POINTS


Reader
Format: Intergraph GeoMedia Access Warehouse
Dataset: ldresses\roadAllowancesAndAddressPoints.mdb
Parameters...
Constraints
Table List:



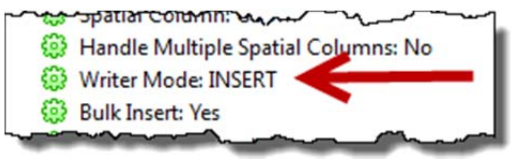
Writer Format	Microsoft SQL Server Spatial
Writer Parameters	Enter the database connection parameters as before.



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.

2) Save and Run Workspace
Click **OK** and then **OK** again to create the workspace.


Spatial Column:
Handle Multiple Spatial Columns: No
Writer Mode: INSERT
Bulk Insert: Yes

Quickly check that the writer parameter 'Writer Mode' is set to INSERT.

Save and then run the workspace. The process takes 5-10 seconds to load 12,292 features.

3) Inspect Output
Use the FME Data Inspector to inspect the contents of the newly written table.
Notice that the attribute PRIMARYINDEX is a unique ID number for each address.

Use the Table View window to view the records.

4) Inspect Updates File
In a text editor open the file *C:\FMEData\Data\Addresses\AddressUpdates\AddressUpdates.txt*
Inspect the contents.

This file contains updates to the main address database. It has almost exactly the same schema, but has an additional field called UPDATE_TYPE, which records the type of edit to carry out:

I	[I]nsert this record to the database as a new address
D	[D]elete this record from the database as an old address
U	[U]pdate this record as a changed address

If you look at the file *C:\FMEData\Data\Addresses\AddressUpdates\AllAddressUpdates.txt* then there will be an extra value for UPDATE_TYPE:

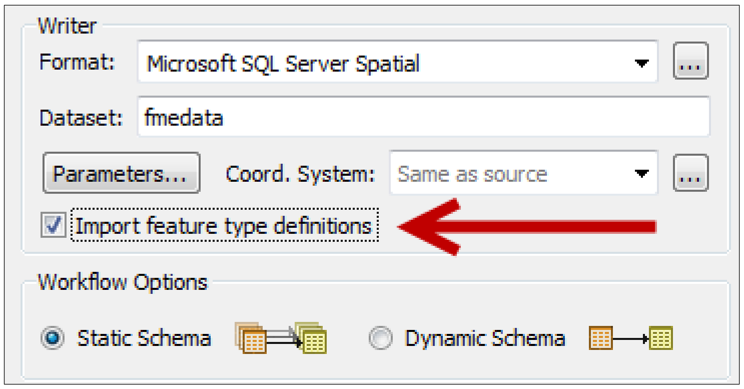
N	This record is u[N]changed so no action is required.

5) Create Updating Workspace
Back in FME Workbench create a new workspace to translate the updates file into SQL Server.

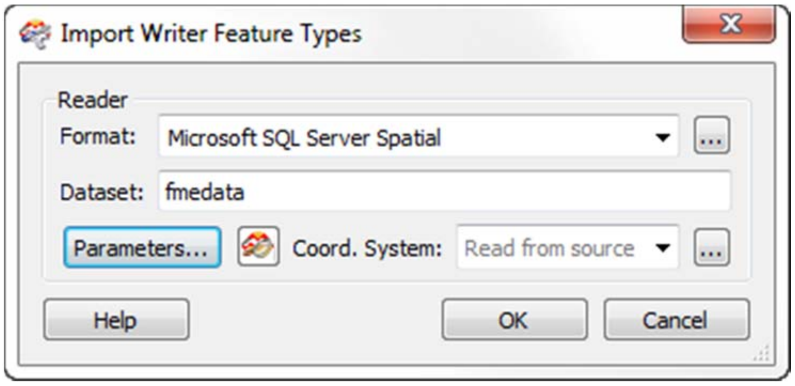
Reader Format	Comma Separated Value (CSV)
Reader Dataset	*C:\FMEData\Data\Addresses\AddressUpdates\AddressUpdates.txt*
Reader Parameters	File Has Field Names = Yes
	Lines to Skip (Header) = 1
Writer Format	SQL Server Spatial

Check the option to "Import Feature Type Definitions" and click **OK**.

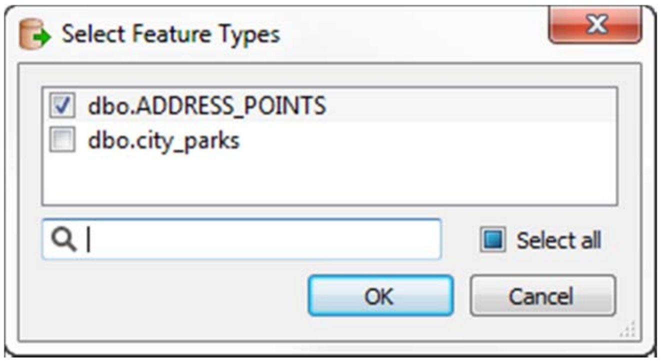
This import option will allow us to use the schema of the database table we just created with the prior workspace.


Writer
Format: Microsoft SQL Server Spatial
Dataset: fmedata
Coord. System: Same as source
[✓] Import feature type definitions
Workflow Options
[✓ Static Schema] Dynamic Schema

6) Select Source of Feature Types
When prompted, set the format and dataset where the feature types can be imported from. By default this should be the SQL Server database, so the fields may not need much editing:


Import Writer Feature Types

Reader
Format: Microsoft SQL Server Spatial
Dataset: fmedata
Coord. System: Read from source

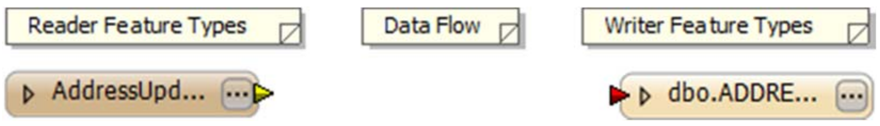

Select Feature Types

☑ dbo.ADDRESS_POINTS
☐ dbo.city_parks

Click the Parameters button and enter the connection parameters (if necessary).

Use the table selection tool to select the newly created ADDRESS_POINTS table.

Click **OK** and then **OK** again. A workspace will be created that looks like this:



Notice that the Writer Feature Type is a copy of the existing database table schema (and not the Reader Feature Type, as a standard workspace would create).

7) Assign Operation Type

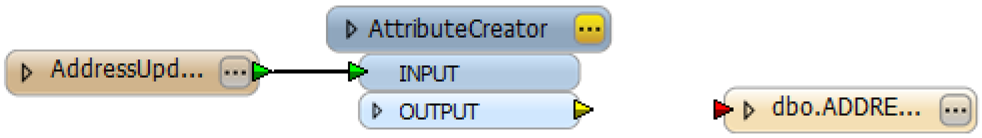
The different action types defined in the updates file need to have different values for *fme_db_operation* in order to carry out the different action for each.

The workspace must be set up to assign the following:

Update Type	fme_db_operation
I	INSERT
D	DELETE
U	UPDATE
N	<none>

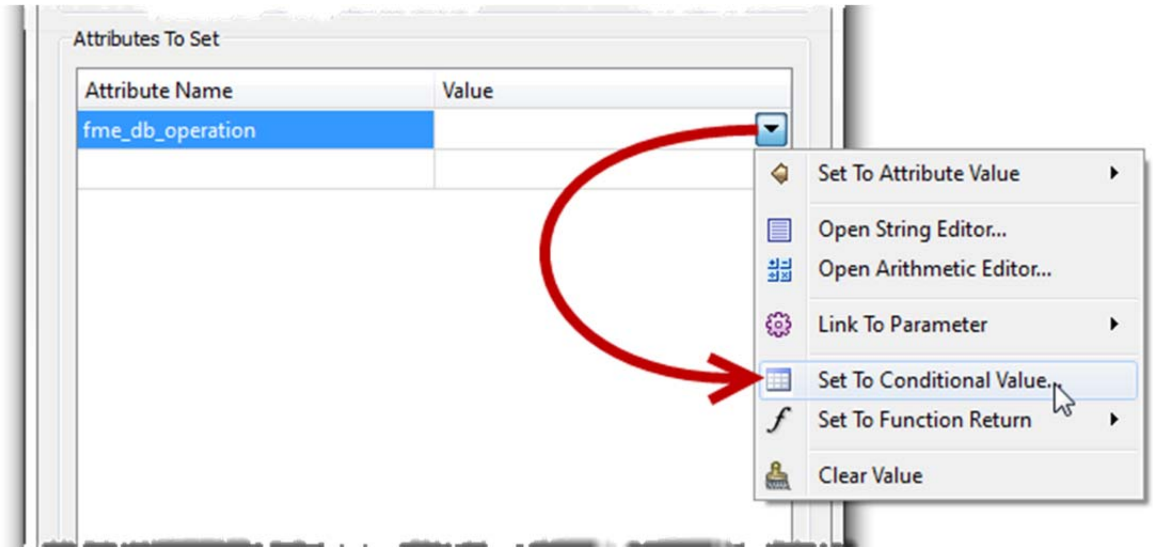
In FME2013-SP1 (or newer) this can be done with an AttributeCreator transformer using new Conditional Mapping functions.

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

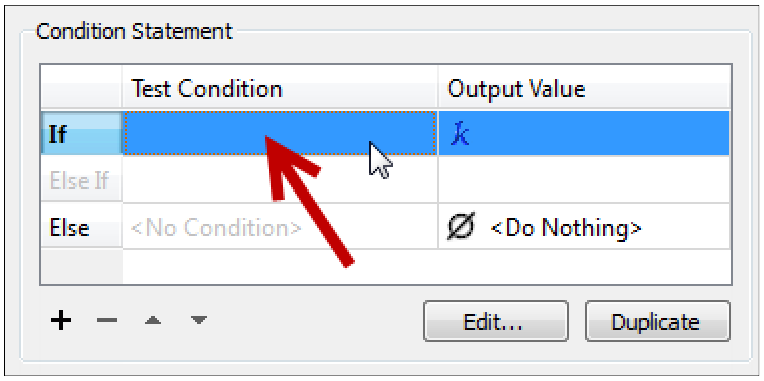


8) Set up INSERT

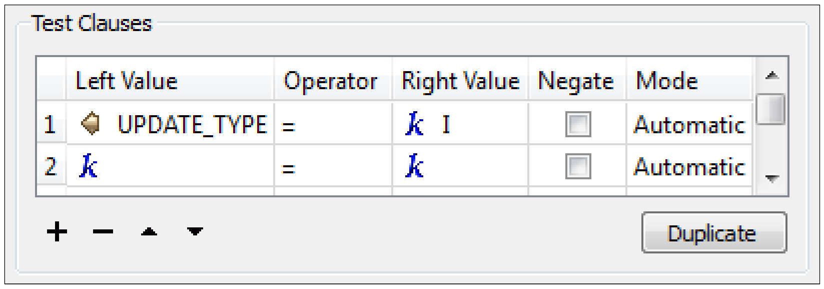
Open the AttributeCreator parameters dialog. Enter fme_db_operation as the Attribute Name to be created. Open the Value drop-down menu and choose “Set to Conditional Value”



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.


Condition Statement

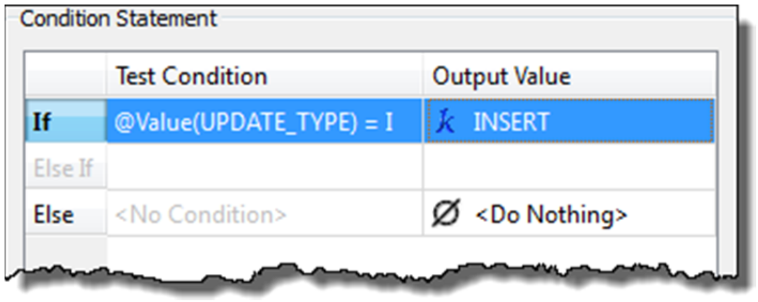
	Test Condition	Output Value
If		
Else If		
Else	<No Condition>	<Do Nothing>


Test Clauses

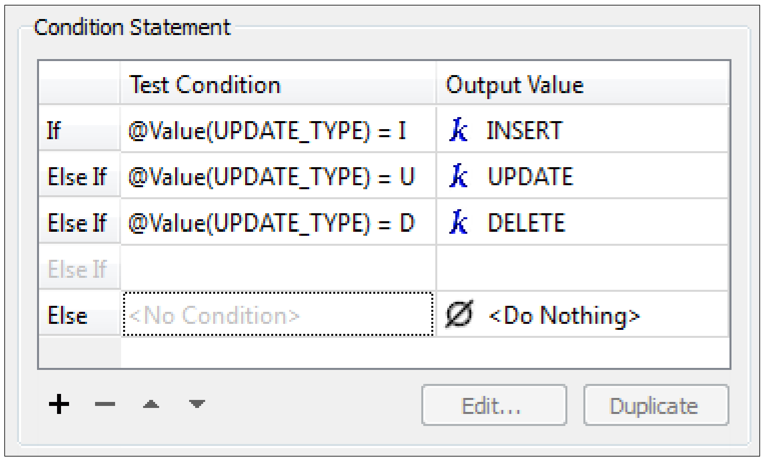
	Left Value	Operator	Right Value	Negate	Mode
1	UPDATE_TYPE	=	I	☐	Automatic
2		=		☐	Automatic

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


Condition Statement

	Test Condition	Output Value
If	@Value(UPDATE_TYPE) = I	INSERT
Else If		
Else	<No Condition>	<Do Nothing>

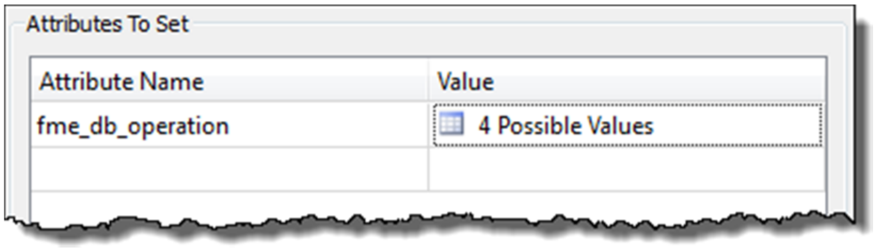

Condition Statement

	Test Condition	Output Value
If	@Value(UPDATE_TYPE) = I	INSERT
Else If	@Value(UPDATE_TYPE) = U	UPDATE
Else If	@Value(UPDATE_TYPE) = D	DELETE
Else If		
Else	<No Condition>	<Do Nothing>

9) Set up UPDATE/DELETE
Now repeat this process to set up Conditional Mapping for the U (UPDATE) and D (DELETE) update types.

10) Assign WHERE Clauses

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


Attributes To Set

Attribute Name	Value
fme_db_operation	4 Possible Values

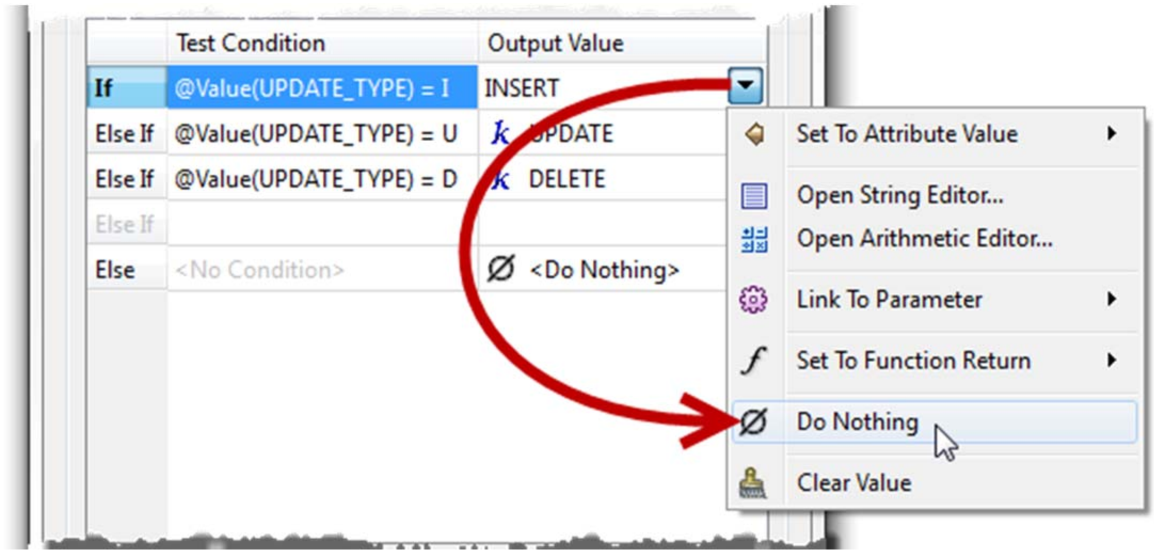
Now we can set up a WHERE clause by creating fme_where.

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”.

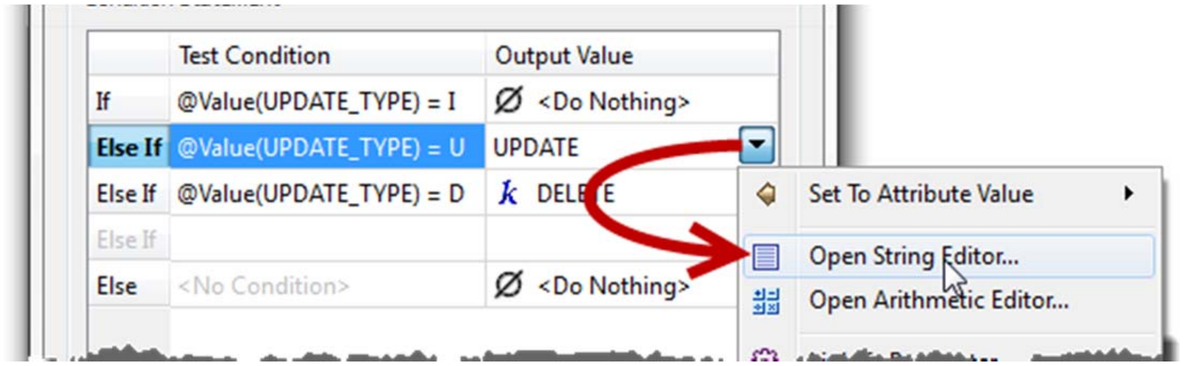
Because this is a set of duplicate conditions, in the Condition Definition dialog we can leave the Test Conditions to be the same and now only need to start editing the output values.

Where UPDATE_TYPE is “I”, 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”:



	Test Condition	Output Value
If	@Value(UPDATE_TYPE) = I	INSERT
Else If	@Value(UPDATE_TYPE) = U	UPDATE
Else If	@Value(UPDATE_TYPE) = D	DELETE
Else If		
Else	<No Condition>	<Do Nothing>

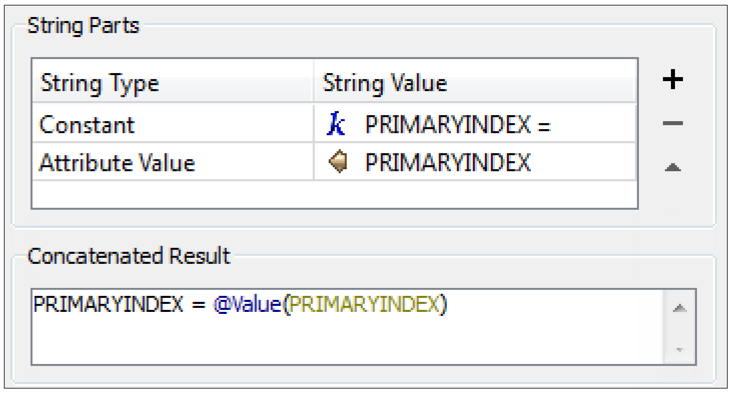
Where UPDATE_TYPE is “U” the output value needs to be a SQL WHERE clause. So click the drop-down menu and select Open String Editor.



	Test Condition	Output Value
If	@Value(UPDATE_TYPE) = I	<Do Nothing>
Else If	@Value(UPDATE_TYPE) = U	UPDATE
Else If	@Value(UPDATE_TYPE) = D	DELETE
Else If		
Else	<No Condition>	<Do Nothing>

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

In the basic editor it will look something like this:



String Parts

String Type	String Value
Constant	PRIMARYINDEX =
Attribute Value	PRIMARYINDEX

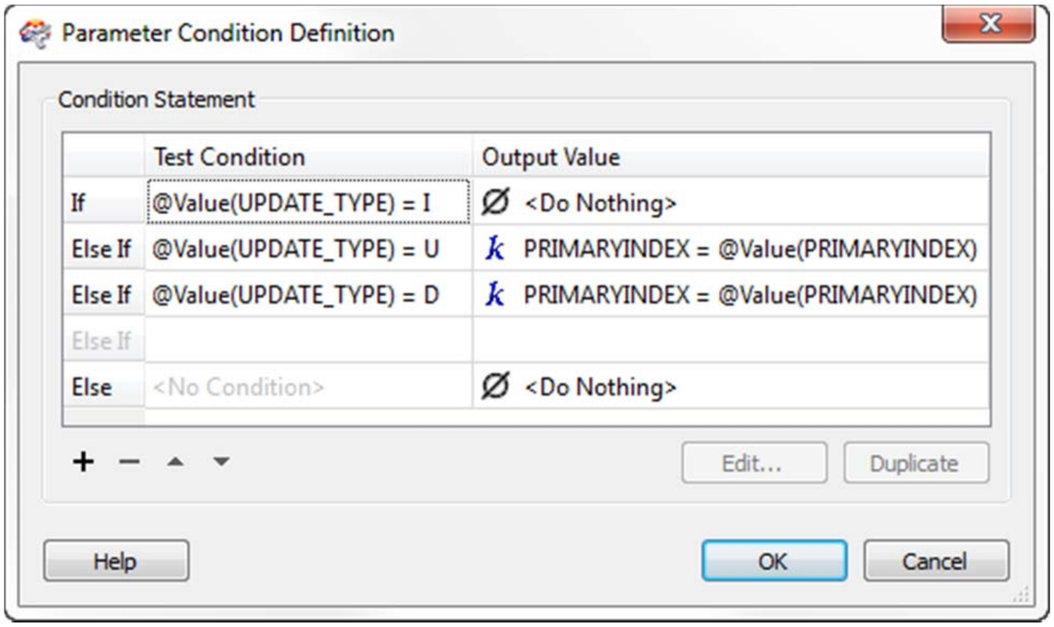
Concatenated Result

```
PRIMARYINDEX = @Value(PRIMARYINDEX)
```

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

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

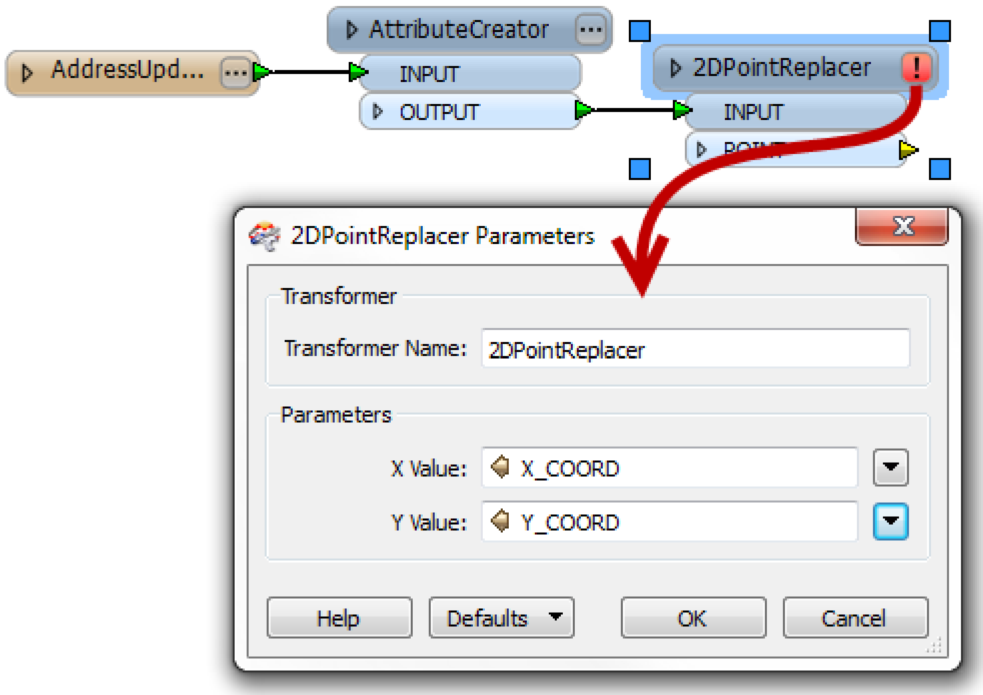
You could, of course, delete the existing DELETE condition and duplicate the UPDATE one.



	Test Condition	Output Value
If	@Value(UPDATE_TYPE) = I	<Do Nothing>
Else If	@Value(UPDATE_TYPE) = U	PRIMARYINDEX = @Value(PRIMARYINDEX)
Else If	@Value(UPDATE_TYPE) = D	PRIMARYINDEX = @Value(PRIMARYINDEX)
Else If		
Else	<No Condition>	<Do Nothing>

11) Create Point Features
The attribute mapping is now complete. However, records from the CSV dataset are just plain text (Non Geometry) features. If they are to be inserted into the database then they must be converted to proper spatial point features.

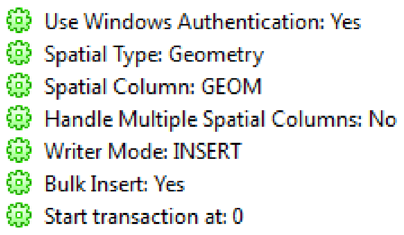
Add a *2DPointReplacer* transformer, connected between the AttributeCreator and Writer. Use the attributes X_COORD and Y_COORD to provide the X/Y coordinate information:



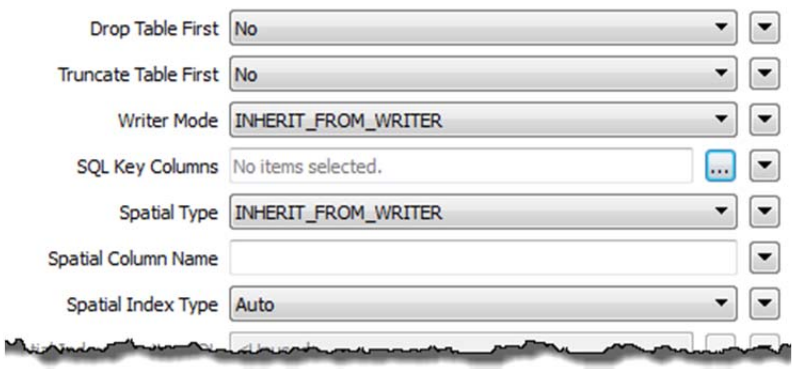
Transformer Name: 2DPointReplacer
X Value: X_COORD
Y Value: Y_COORD

12) 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



Drop Table First	No
Truncate Table First	No
Writer Mode	INHERIT_FROM_WRITER
SQL Key Columns	No items selected.
Spatial Type	INHERIT_FROM_WRITER
Spatial Column Name	
Spatial Index Type	Auto

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

Because we imported the feature type, FME knows the table exists and so sets “Drop Table” to No. That’s fine for us, and we don’t want to truncate the contents either.

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:

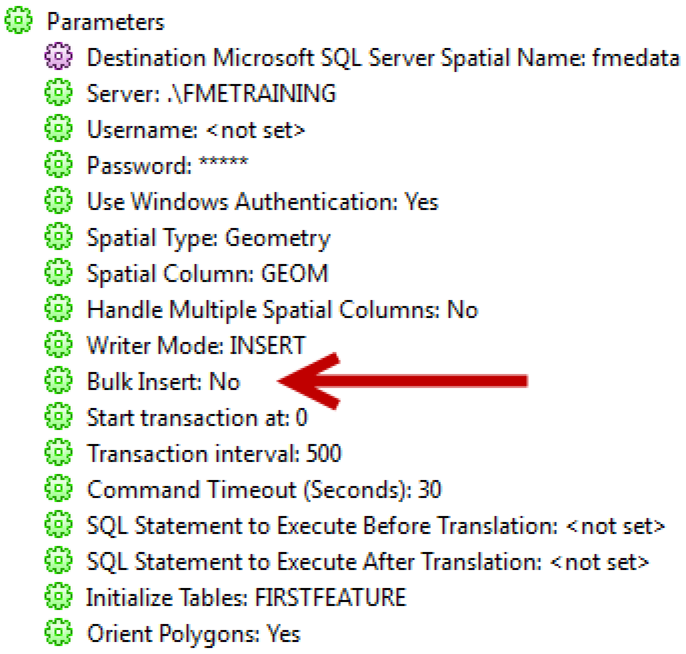
- Leave the writer:writer mode parameter as-is, and set the feature type:writer mode parameter to UPDATE.
- Set the writer:writer mode parameter to UPDATE, and leave the feature type:writer mode parameter as-is.

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

13) Save and Run Workspace
Save the workspace, 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.


Parameters
Destination Microsoft SQL Server Spatial Name: fmedata
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

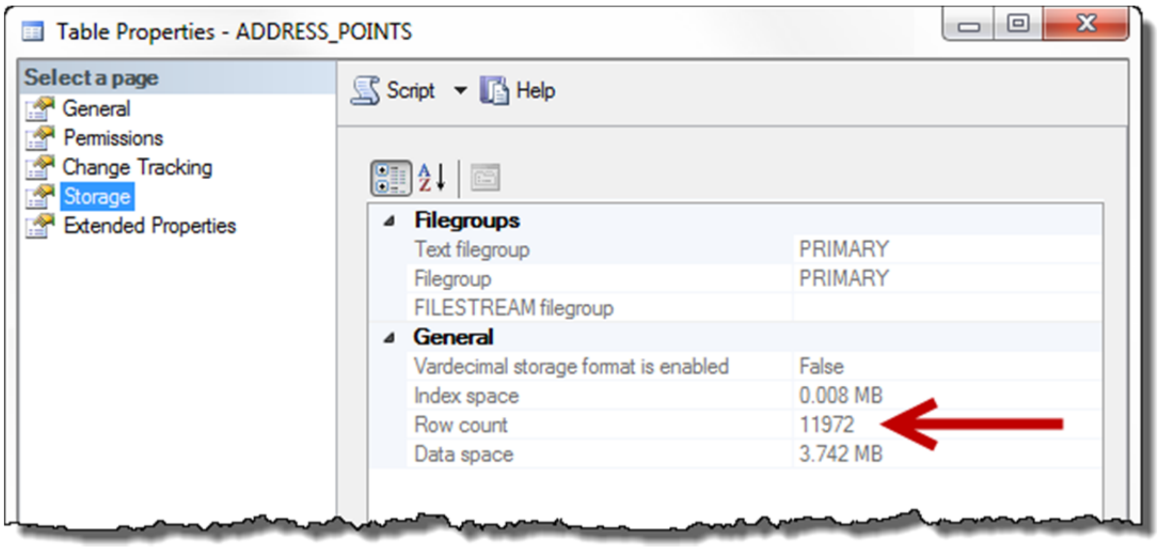
14) Run Workspace
Run the workspace once more.

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 1,465 features are written to the database. These are composed of:

INSERTS	284
DELETES	604
UPDATES	577

Because ORIGINAL features (12,292) + INSERTS (284) – DELETES (604) = 11972, there should now be 11,972 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	11972
Data space	3.742 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.

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.

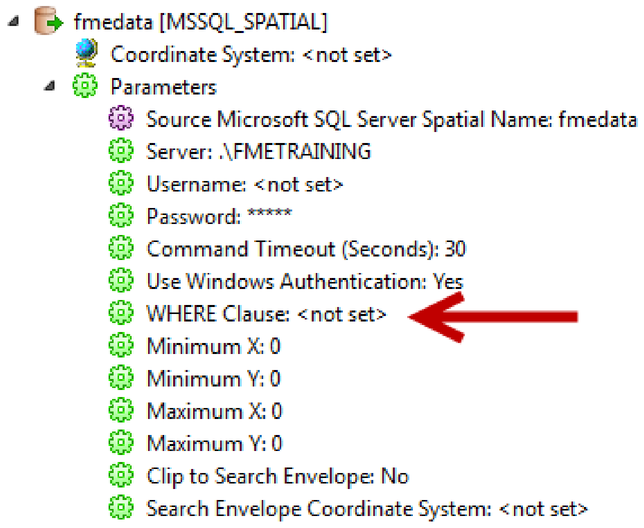
Like a restaurant, it's very expensive and time-consuming to order all data from a database just to discard most of it. Far better to order only the data you intend to consume in the current session!'

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

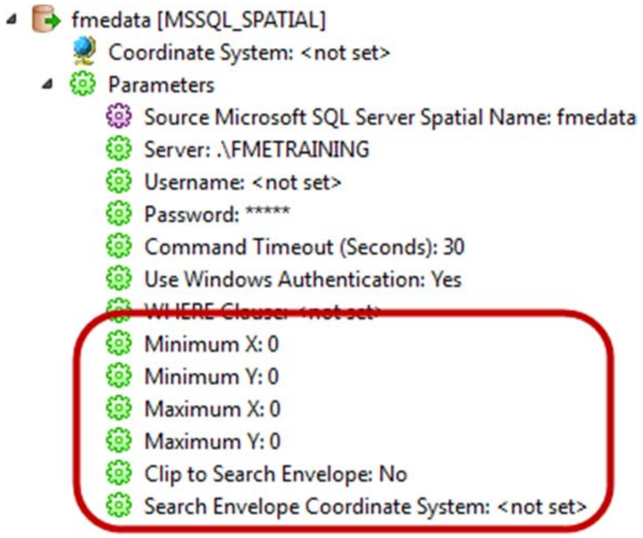
Where Clause
Most database readers will have a "where clause" parameter. Here a query can be set, so that only features that pass the query will be returned to FME.

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.



fmedata [MSSQL_SPATIAL]
Coordinate System: <not set>
Parameters
Source Microsoft SQL Server Spatial Name: fmedata
Server: .\FMETRAINING
Username: <not set>
Password: *****
Command Timeout (Seconds): 30
Use Windows Authentication: Yes
WHERE Clause: <not set>
Minimum X: 0
Minimum Y: 0
Maximum X: 0
Maximum Y: 0
Clip to Search Envelope: No
Search Envelope Coordinate System: <not set>

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.



fmedata [MSSQL_SPATIAL]
Coordinate System: <not set>
Parameters
Source Microsoft SQL Server Spatial Name: fmedata
Server: .\FMETRAINING
Username: <not set>
Password: *****
Command Timeout (Seconds): 30
Use Windows Authentication: Yes
WHERE Clause: <not set>
Minimum X: 0
Minimum Y: 0
Maximum X: 0
Maximum Y: 0
Clip to Search Envelope: No
Search Envelope Coordinate System: <not set>

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 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 3: Where Clause	
Scenario	FME user; City of Interopolis, Planning Department
Data	SQL Server (Address Point Data)
Overall Goal	Read address data with a query
Demonstrates	Where Clause parameters
Finished Workspace	C:\FMEData\Workspaces\PathwayManuals\Database3(MSSQL)-Complete.fmw

To test the previous example, it's time to read back some data – but only for a specific zipcode.

1) Start FME Workbench
Start Workbench if necessary and begin with an empty workspace.

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

Reader Format	SQL Server Spatial

Click the reader parameters button and enter the database connection parameters as before. Click the Table List button and select the ADDRESS_POINTS table.

Click **OK**, and **OK** again to close the dialogs and add the new reader.

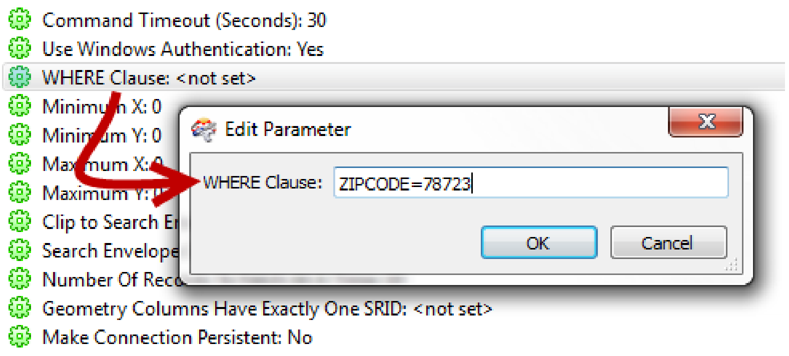
3) Add Inspector
Connect an Inspector transformer to the reader feature type.

4) Save and Run Workspace
Save and run the workspace. Note how many features were read and how long it took. On my computer it reads 11,972 features in 2.6 seconds.

5) Set Where Clause
Locate the WHERE Clause parameter in the Navigator Window, and double-click it.

In the Edit Parameter box, enter:

```
ZIPCODE=78723
```


Command Timeout (Seconds): 30
Use Windows Authentication: Yes
WHERE Clause: <not set>
Minimum X: 0
Minimum Y: 0
Maximum X: 0
Maximum Y: 0
Clip to Search E
Search Envelope
Number Of Reco
Geometry Columns Have Exactly One SRID: <not set>
Make Connection Persistent: No

Edit Parameter
WHERE Clause: ZIPCODE=78723

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 6,584 features in 1.9 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.

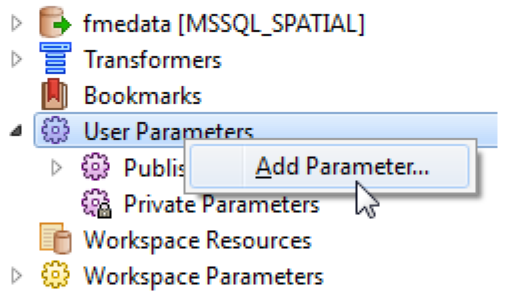
***NB**: Scripted Parameters do a similar task, but for more complex scenarios where the value has to be incorporated using a Python or Tcl script. This would be more useful for manipulating the search envelope parameter values.*

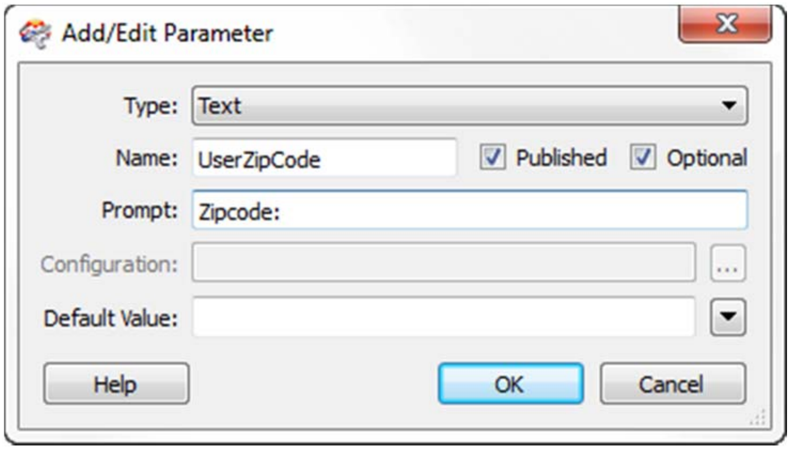
Example 4: Concatenated Parameter	
Scenario	FME user; City of Interopolis, Planning Department
Data	SQL Server (Address Point Data)
Overall Goal	Read address data with a user-defined query
Demonstrates	Concatenated parameters for a database
Starting Workspace	C:\FMEData\Workspaces\PathwayManuals\Database4(MSSQL)-Begin.fmw
Finished Workspace	C:\FMEData\Workspaces\PathwayManuals\Database4a(MSSQL)-Complete.fmw C:\FMEData\Workspaces\PathwayManuals\Database4b(MSSQL)-Complete.fmw

Here we wish to let the user choose which zipcode to read data from.

1) Start FME Workbench
Start Workbench (if necessary) and open the workspace from the previous example. Alternatively you can open *C:\FMEData\Workspaces\PathwayManuals\Database4(MSSQL)-Begin.fmw*

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




Add/Edit Parameter

Type: Text
Name: UserZipCode [✓ Published] [✓ Optional]
Prompt: Zipcode:
Configuration:
Default Value:

When prompted, choose a parameter of type text. Set the name to UserZipCode and the prompt to ZipCode. Click OK to create the parameter.

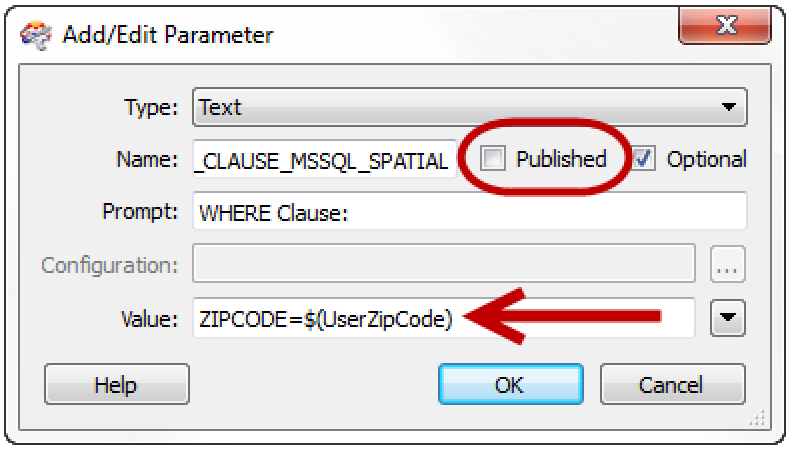
This creates the parameter for the end user to select the zipcode.

3) Add Parameter

Right-click the Where Clause reader parameter and choose Create User Parameter.
Leave the type as Text, and set the Where clause default value to:

ZIPCODE=$(UserZipCode)

The easiest method is to use the Text Editor where the "ZIPCODE=" part can be typed manually (as a constant) and the published parameter can be selected from a list.



Type: Text
Name: _CLAUSE_MSSQL_SPATIAL
Published [ ]
Optional [✓]
Prompt: WHERE Clause:
Configuration:
Value: ZIPCODE=$(UserZipCode)

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

Run the workspace using prompt and run.
When prompted, enter a zipcode into the field provided.

Valid values are:

- 78722
- 78724
- 78752
- 78754
- 78723
- 78751
- 78753

Each different zipcode will change the database where clause and return a different set of data.

Advanced Task

If you have a little time, recreate the *UserZipCode* parameter to be a choice, rather than a text parameter. The choices should reflect the above list of valid zipcodes for Interopolis.

But if you did have to find the list of unique zipcodes in the data, how would you do it?!

Mr. Saif-Investor says…

'A WHERE clause on the reader applies the same query to ALL tables being read. This can be useful, but also inefficient.

Be aware that there is also a similar WHERE clause parameter on each feature type, so data can be queried only on individual tables.'

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 5: FeatureReader	
Scenario	FME user; City of Interopolis, Planning Department
Data	Census Data (ESRI Shape) Address Point Data (SQL Server)
Overall Goal	Read address data within a user-defined census area
Demonstrates	FeatureReader Transformer
Finished Workspace	C:\FMEData\Workspaces\PathwayManuals\Database5(MSSQL)-Complete.fmw

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

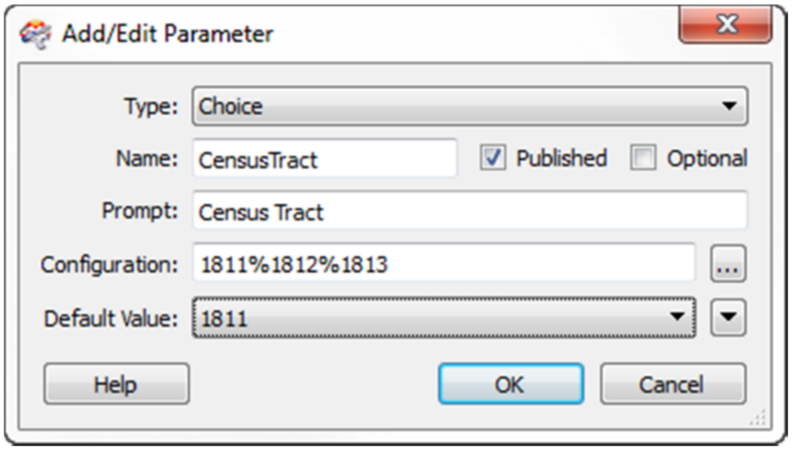
1) Start FME Workbench
Start Workbench if necessary and begin with an empty workspace.

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

Reader Format	ESRI Shape
Reader Dataset	*C:\FMEData\Data\GovtBoundaries\TravisCounty\CensusTracts.shp*

This is the dataset that contains the census tract boundaries.

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


Add/Edit Parameter

Type: Choice
Name: CensusTract ☑ Published ☐ Optional
Prompt: Census Tract
Configuration: 1811%1812%1813
Default Value: 1811

When prompted, choose a parameter of type Choice. Set the name to CensusTract and the prompt to Census Tract. Click OK to create the parameter.

In the configuration dialog choose values 1811, 1812, 1813. Set the Default Value field to 1811

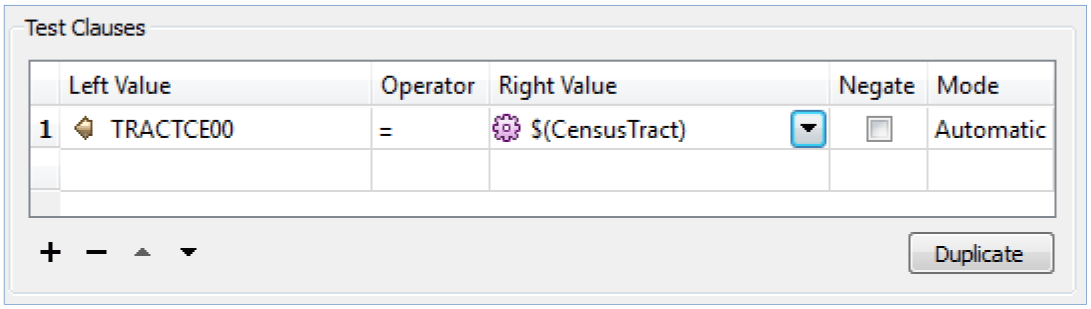
The optional flag should be unset.

4) Add Tester

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

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

Left Value	Attribute Value > TRACTCE00
Operator	=
Right Value	Parameter > Census Tract


Test Clauses

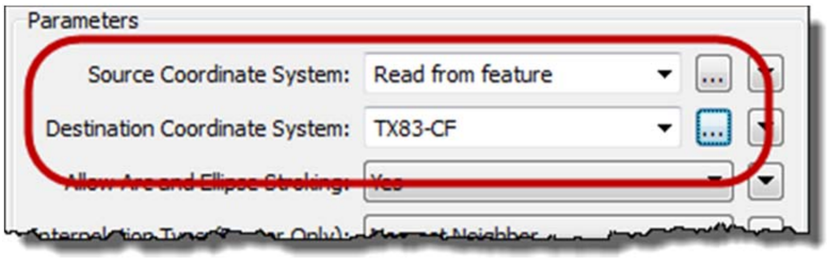
	Left Value	Operator	Right Value	Negate	Mode
1	TRACTCE00	=	$(CensusTract)	☐	Automatic

5) Add Reprojector

The Shape data is in a Lat/Long coordinate system, but the database data is in TX83-CF. If a spatial operation is to be carried out successfully the Shape data must be reprojected to match.

Add a *Reprojector* transformer connected to the Tester:PASSED port. Open the properties dialog. Set the following parameters:

Source Coordinate System	Read from feature
Destination Coordinate System	TX83-CF


Parameters

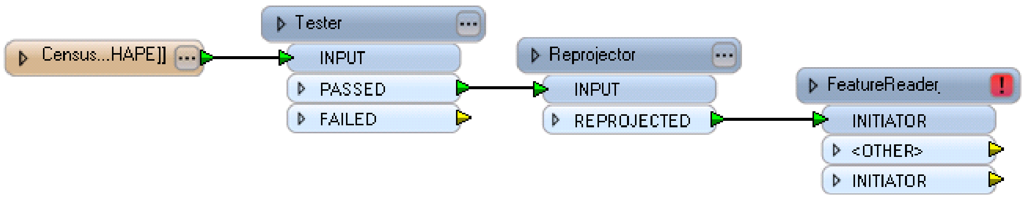
Source Coordinate System: Read from feature

Destination Coordinate System: TX83-CF

Allow Arc and Ellipse Stroking: Yes

6) Add FeatureReader

Add a *FeatureReader* transformer connected to the Reprojector:REPROJECTED port. 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. Click the Table List button and select the ADDRESS_POINTS table.

Panel 2:

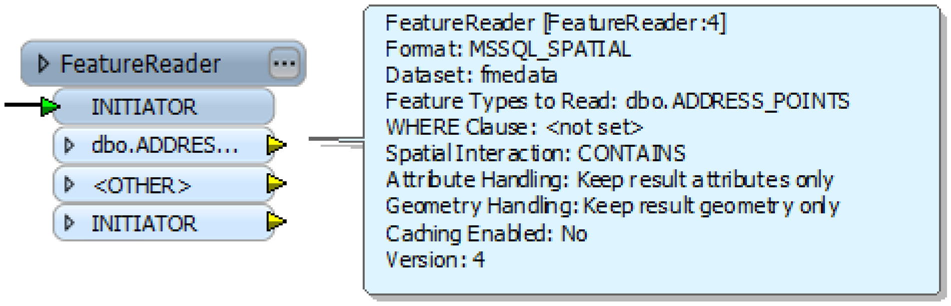
Feature Types	Query the Feature Types specified on the previous page
Where Clause	<none>

Panel 3:

Spatial Test	CONTAINS

Panel 4:

Attribute Handling	Result Attributes Only
Geometry Handling	Result Geometry Only



FeatureReader [FeatureReader:4]
Format: MSSQL_SPATIAL
Dataset: fmedata
Feature Types to Read: dbo.ADDRESS_POINTS
WHERE Clause: <not set>
Spatial Interaction: CONTAINS
Attribute Handling: Keep result attributes only
Geometry Handling: Keep result geometry only
Caching Enabled: No
Version: 4

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 a census tract of your choice.

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

The data for tract 001811 will look like this:

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 <u>in that table</u> (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.

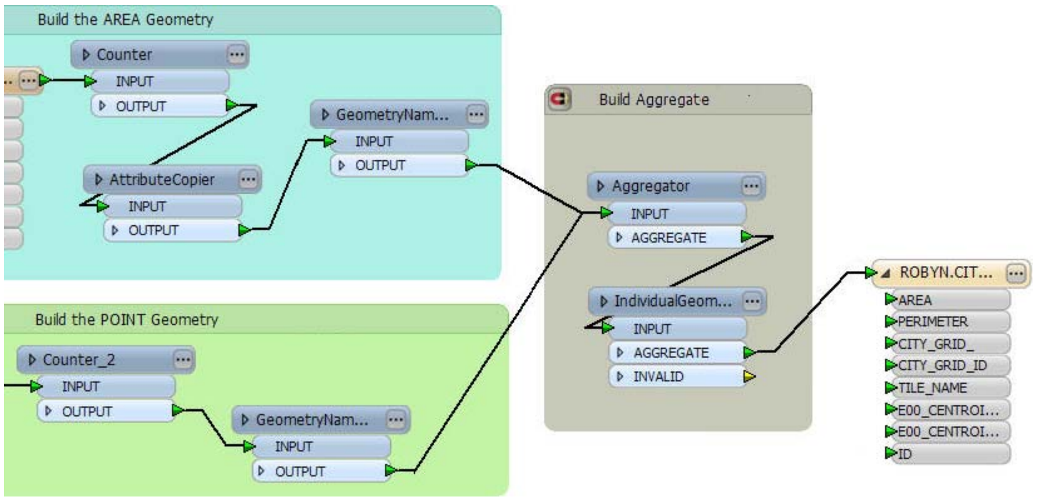
The functionality used to do this involves geometry names and aggregates.

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.

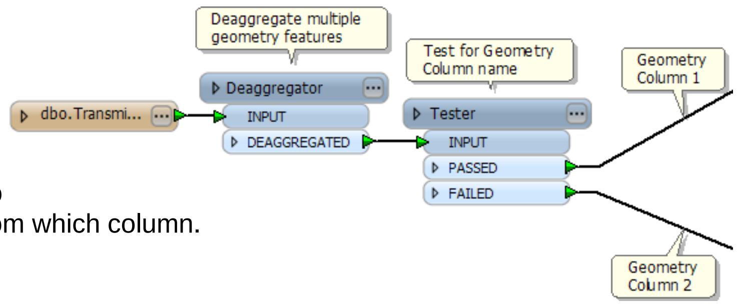
The features are grouped together as an aggregate - usually with an *Aggregator* transformer – and this identifies which features form a particular database record.

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; City of Interopolis, Planning Department
Data	City Parks (MapInfo TAB, SQL Server)
Overall Goal	Write to multiple geometry columns
Demonstrates	Multiple Geometry Writing
Finished Workspace	C:\FMEData\Workspaces\PathwayManuals\Database6(MSSQL)-Complete.fmw
SQL Script	C:\FMEData\Workspaces\PathwayManuals\Database6(MSSQL)-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 (MFAL)
Reader Dataset	*C:\FMEData\Data\Parks\city_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:

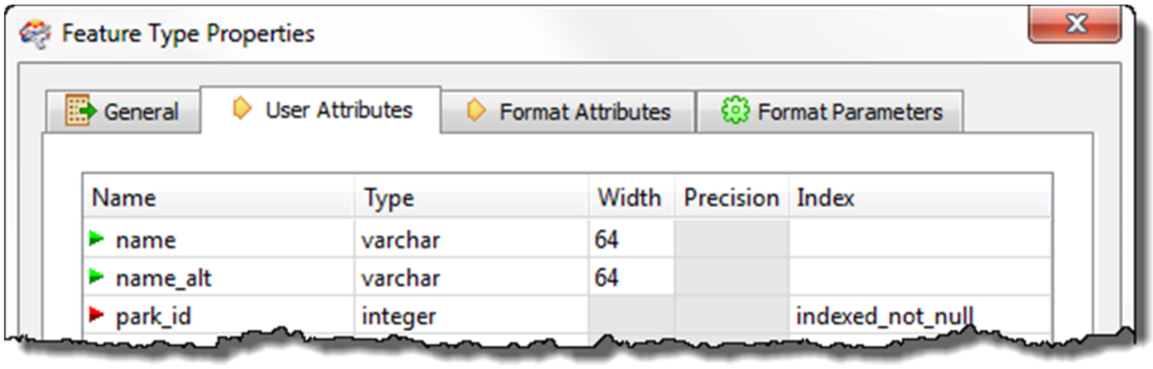
```
FME_SQL_DELIMITER ;
IF  EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[dbo].[city_parks]') AND type in (N'U'))
DROP TABLE [dbo].[city_parks];

CREATE TABLE [dbo].[city_parks](
       [park_id] [int] NOT NULL,
       [name] [varchar](64) NULL,
       [name_alt] [varchar]64) NULL,
       [Parks_Points] [geometry] NULL,
        [Parks_Polygons] [geometry] NULL
);
```

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) Edit Schema
Notice (in the script) that the new table has a column *park_id* which is not defined in the workspace. Edit the destination feature type and add the new field.

Set the indexing type to *indexed_not_null* (if there is going to be an ID, it may as well be indexed)

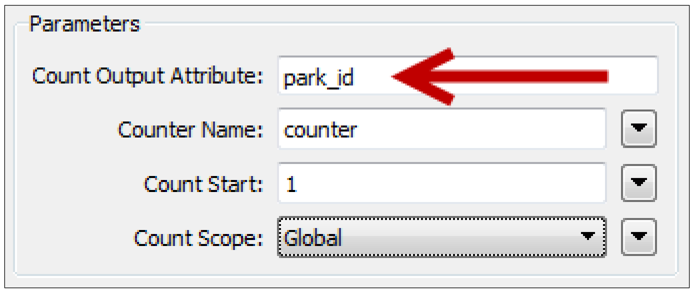


Name	Type	Width	Precision	Index
name	varchar	64		
name_alt	varchar	64		
park_id	integer			indexed_not_null

4) Add Counter
To create a park ID, add a *Counter* transformer.

Open the properties dialog and make Count Output Attribute write to *park_id*

Change the Count Start parameter to start at 1 (instead of 0)

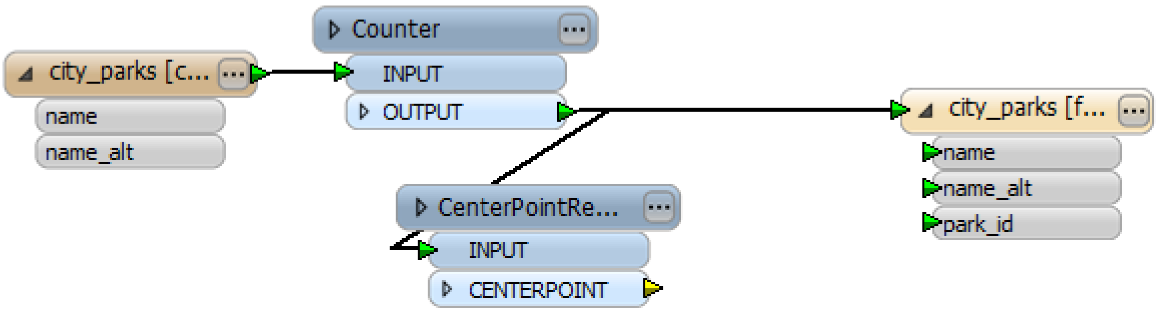


Count Output Attribute: park_id
Counter Name: counter
Count Start: 1
Count Scope: Global

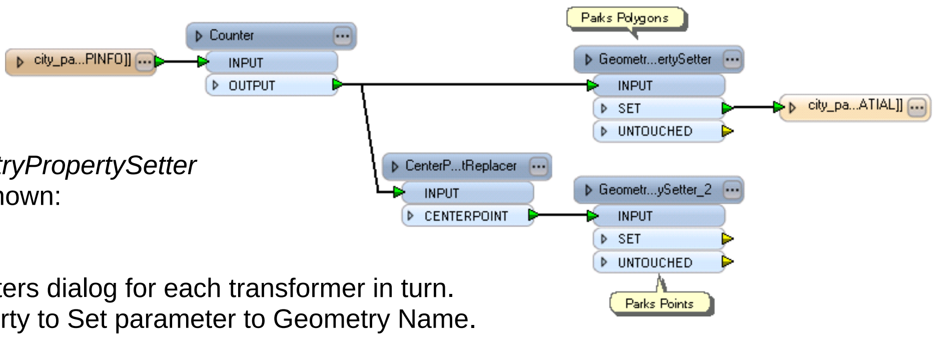
5) Add CenterPointReplacer
Now to create the multiple geometries: The park features are originally polygons; to create points insert a *CenterPointReplacer* transformer, connected to a second output stream from the Counter.

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

The workspace will now look something like this:



6) 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: Parks_Points and Parks_Polygons



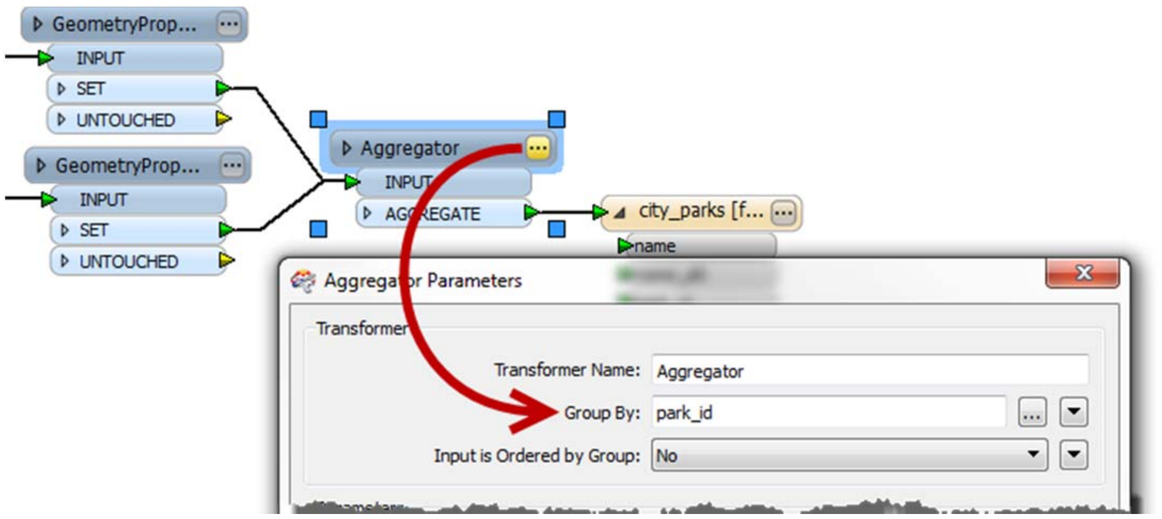
Place two *GeometryPropertySetter* transformers as shown:

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 Parks_Polygons or Parks_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.

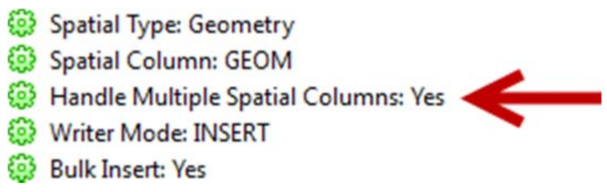
In the parameters, set *group_by* to use *park_id*; this is how the two sets of features are paired off and aggregated together.



Transformer Name: Aggregator
Group By: park_id
Input is Ordered by Group: No

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 W riter parameter *Handle Multiple Spatial Columns*, and set it to Yes.

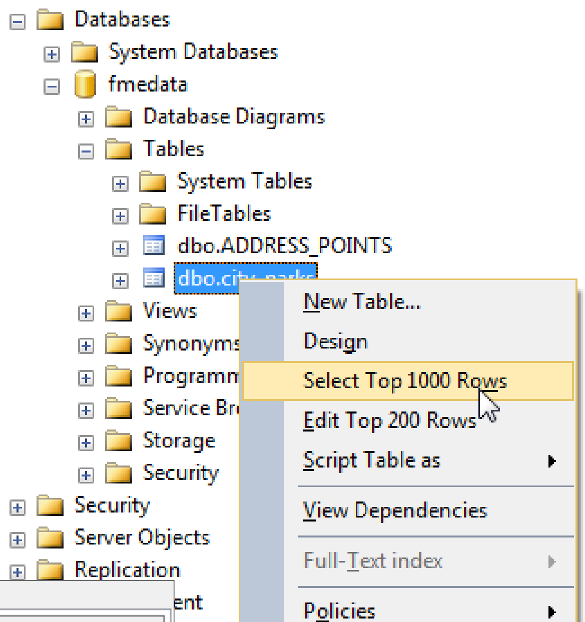

Spatial Type: Geometry
Spatial Column: GEOM
Handle Multiple Spatial Columns: Yes
Writer Mode: INSERT
Bulk Insert: Yes

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.

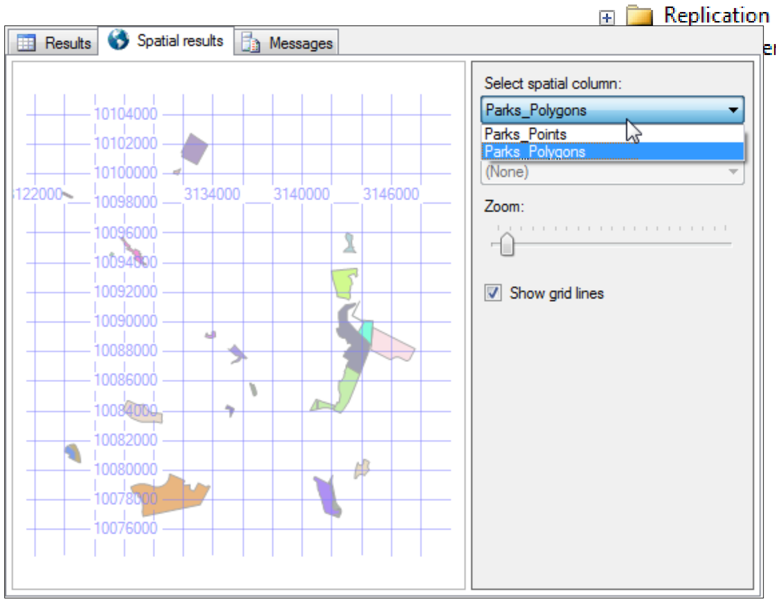
11) Browse for Database
In the Object Explorer window, browse to the Interopolis database, and then browse to the city_parks table.

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; City of Interopolis, Planning Department
Data	City Parks (SQL Server, KML)
Overall Goal	Read and filter multiple geometry columns
Demonstrates	Multiple Geometry Reading
Finished Workspace	C:\FMEData\Workspaces\PathwayManuals\Database7(MSSQL)-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 city_parks as the table to read (don't confuse it with any table called CityParks!)

Writer Format	Google Earth KML
Writer Dataset	*C:\FMEData\Output\DemoOutput\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 Deaggregator
Add a Deaggregator 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 Parks_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.

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

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

The point's version should be set to:

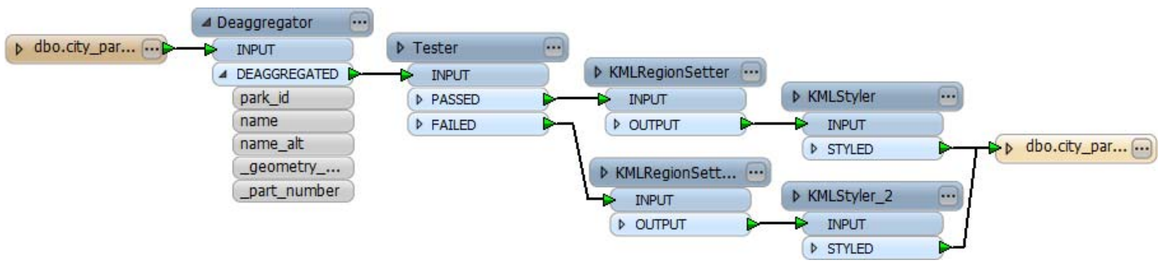
Bounding Box: Calculate:	Yes – 2D
Buffer XY Region By	0.003
Minimum Display Size	10
Maximum Display Size	30

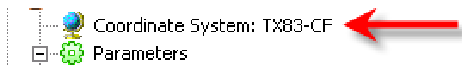
The polygon's version should be set to:

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

6) 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:




Coordinate System: TX83-CF
Parameters

7) Set Coordinate System
Set the source coordinate system for the reader to TX83-CF. The KML writer will complain if we don't!

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.

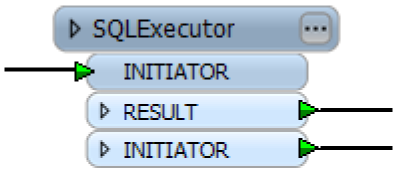
Such statements might be used to:

- 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.



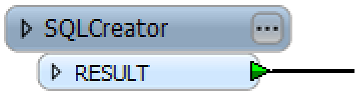
If the SQL is a query, and if features are returned from the database, those features form the output from the transformer. There will be a feature output for each row of the results.

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; City of Interopolis, Planning Department
Data	Address Data (SQL Server) Emergency Facilities (Access Database)
Overall Goal	Carry out a join using the SQLExecutor
Demonstrates	SQLExecutor transformer
Finished Workspace	C:\FMEData\Workspaces\PathwayManuals\Database8(MSSQL)-Complete.fmw

The city has a database of emergency facilities in Microsoft Access format.

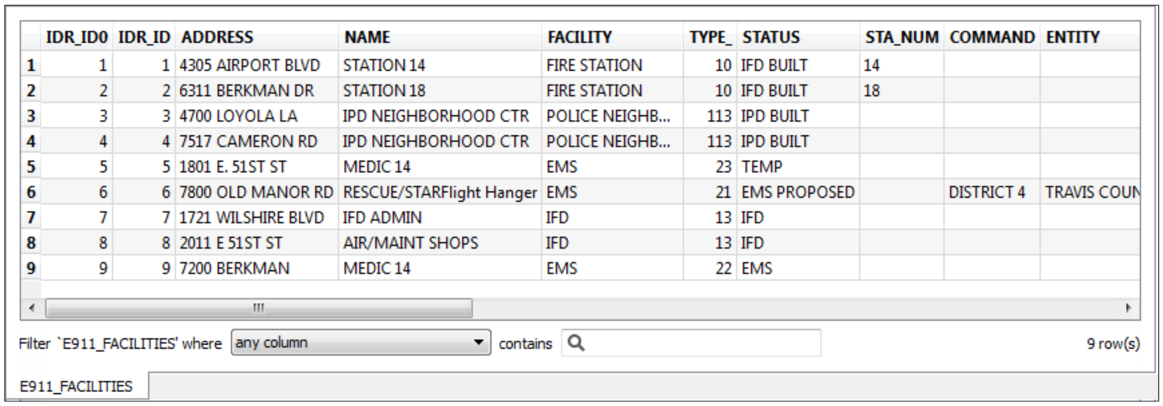
In this example, FME will be used to update the address database, to flag addresses that are an emergency facility. This will be done by using a SQLExecutor transformer to do a database join.

1) Inspect Source Data

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

Reader Format	Microsoft Access
Reader Dataset	*C:\FMEData\Data\Emergency\e911_facilities.mdb*

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



	IDR_ID0	IDR_ID	ADDRESS	NAME	FACILITY	TYPE_	STATUS	STA_NUM	COMMAND	ENTITY
1	1	1	4305 AIRPORT BLVD	STATION 14	FIRE STATION	10	IFD BUILT	14		
2	2	2	6311 BERKMAN DR	STATION 18	FIRE STATION	10	IFD BUILT	18		
3	3	3	4700 LOYOLA LA	IPD NEIGHBORHOOD CTR	POLICE NEIGHB...	113	IPD BUILT			
4	4	4	7517 CAMERON RD	IPD NEIGHBORHOOD CTR	POLICE NEIGHB...	113	IPD BUILT			
5	5	5	1801 E. 51ST ST	MEDIC 14	EMS	23	TEMP			
6	6	6	7800 OLD MANOR RD	RESCUE/STARFlight Hanger	EMS	21	EMS PROPOSED		DISTRICT 4	TRAVIS COUN
7	7	7	1721 WILSHIRE BLVD	IFD ADMIN	IFD	13	IFD			
8	8	8	2011 E 51ST ST	AIR/MAINT SHOPS	IFD	13	IFD			
9	9	9	7200 BERKMAN	MEDIC 14	EMS	22	EMS			

Filter `E911_FACILITIES' where any column contains 9 row(s)

E911_FACILITIES

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	SQL Server Spatial

Click the reader parameters button and enter the database connection parameters as before. Click the Table List button and select the ADDRESS_POINTS table.

Also, because we don't want to work with inactive addresses, set a WHERE Clause:

```
STAT != 'INAC'
```

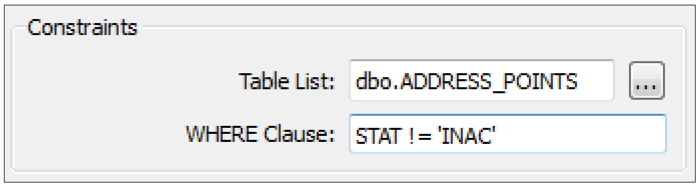


Table List: dbo.ADDRESS_POINTS
WHERE Clause: STAT != 'INAC'

Click **OK**, and **OK** again to close the dialogs and add the new reader.

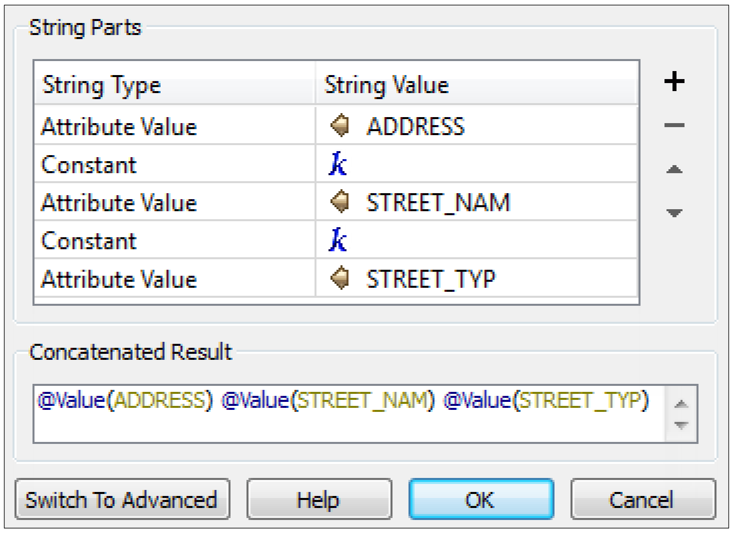
To confirm this works, connect an *Inspector* transformer to the Reader feature type and run the workspace.

4) Add StringConcatenator
If you noticed, the Access dataset stores an address as a single string, rather than a set of separate attributes. To do a join requires that address table attributes should be concatenated to match this structure.

Add an AttributeCreator transformer to the workspace. Open the parameters dialog and add "fulladdr" as the attribute to create. Under Value choose "Open String Editor".

In the String Builder dialog set it up to concatenate:

```
Attribute Value    ADDRESS
Constant           <space character>
Attribute Value    STREET_NAM
Constant           <space character>
Attribute Value    STREET_TYP
```

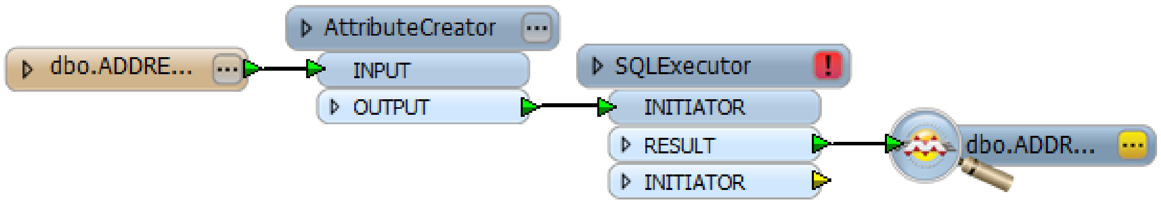


String Type	String Value
Attribute Value	ADDRESS
Constant	
Attribute Value	STREET_NAM
Constant	
Attribute Value	STREET_TYP

@Value(ADDRESS) @Value(STREET_NAM) @Value(STREET_TYP)

Re-run the workspace if desired, to prove that the concatenation is working correctly.

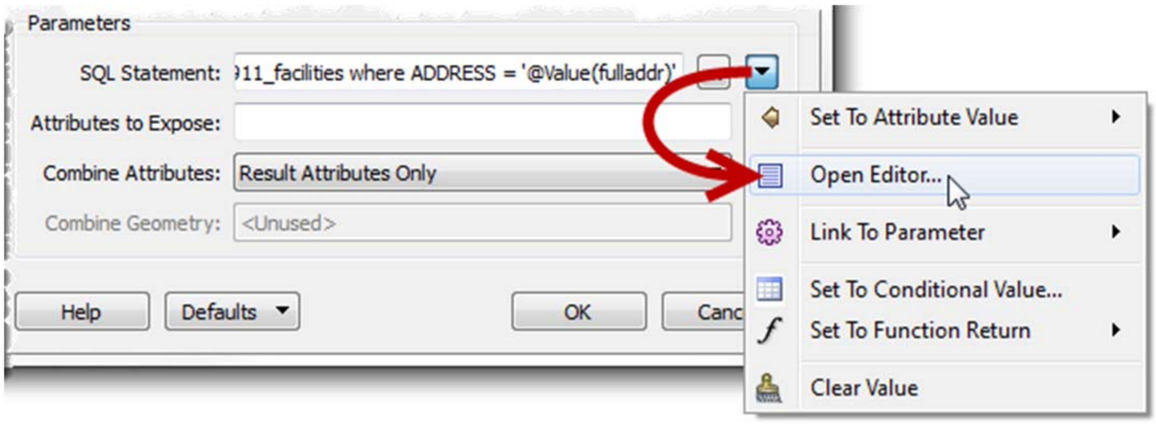
5) Add SQLExecutor
Add a *SQLExecutor* transformer after the *AttributeCreator*.
The RESULT output port is the one to connect to the output.



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

Reader Format	Microsoft Access
Reader Dataset	C:\FMEData\Data\Emergency\e911_facilities.mdb
SQL Statement	select * from e911_facilities where ADDRESS = '@Value(fulladdr)'
Attributes to Expose	NAME
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!


Parameters
SQL Statement: ›11_facilities where ADDRESS = '@Value(fulladdr)'
Attributes to Expose:
Combine Attributes: Result Attributes Only
Combine Geometry: <Unused>

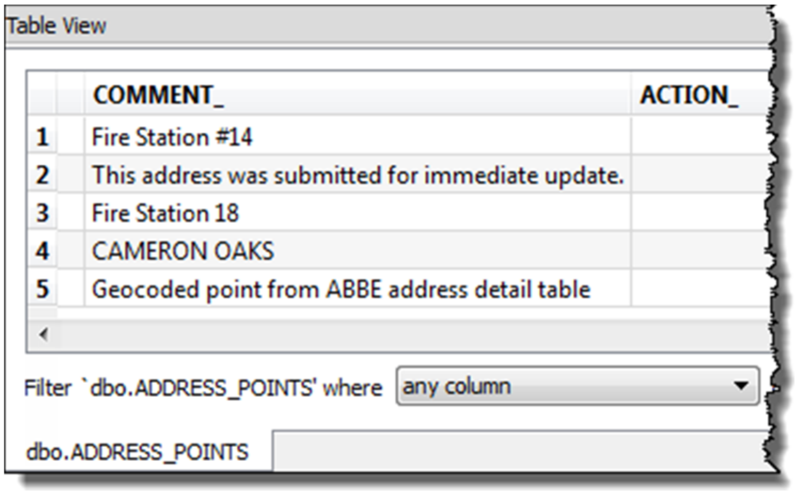
The Attributes to Expose is most easily set by manually entering the attribute name. Although you can also use the option to populate this by querying the database for a list of attributes, in this particular case it's more effort to do so.

"Keep Initiator Attributes if Conflict" ensures we're keeping the NAME field from the Emergency Services (Access) dataset, and not any NAME field from the SQL Server Address table.

7) Add AttributeCopier
Add an AttributeCopier transformer to copy *NAME* to *COMMENT_*

COMMENT_ 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 emergency facility in the comments field of the address table.



	COMMENT_	ACTION_
1	Fire Station #14	
2	This address was submitted for immediate update.	
3	Fire Station 18	
4	CAMERON OAKS	
5	Geocoded point from ABBE address detail table	

Filter `dbo.ADDRESS_POINTS' where any column

8) Save and Run Workspace
Save the workspace and then run it. Inspect the output to confirm the COMMENT_ field now includes the contents of the emergency data 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 addresses that are emergency facilities.

Geometry

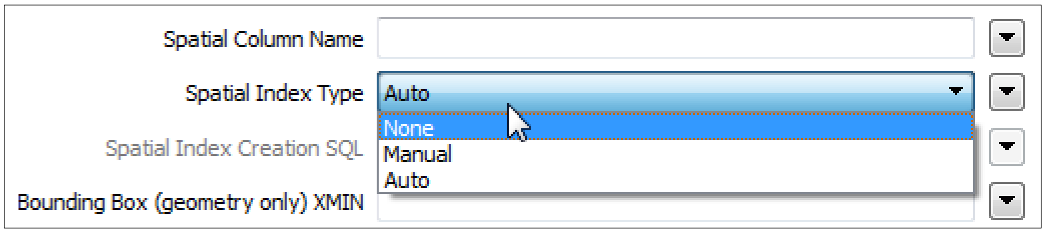
Like any spatial format, it's important to be aware of what geometries are supported within SQL Server and how to clean bad geometries to prevent translation failures.

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".



Spatial Column Name

Spatial Index Type: Auto

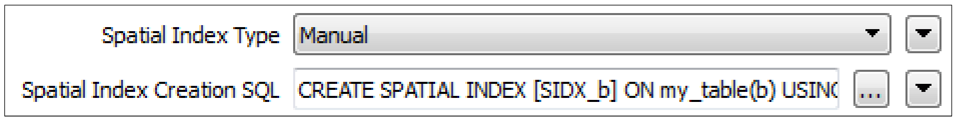
Spatial Index Creation SQL

Bounding Box (geometry only) XMIN

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:

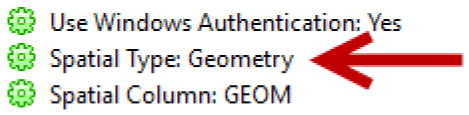


Spatial Index Type: Manual

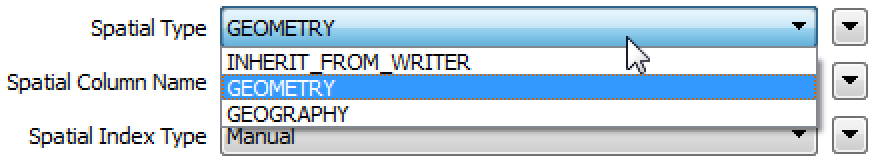
Spatial Index Creation SQL: CREATE SPATIAL INDEX [SIDX_b] ON my_table(b) USINC

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

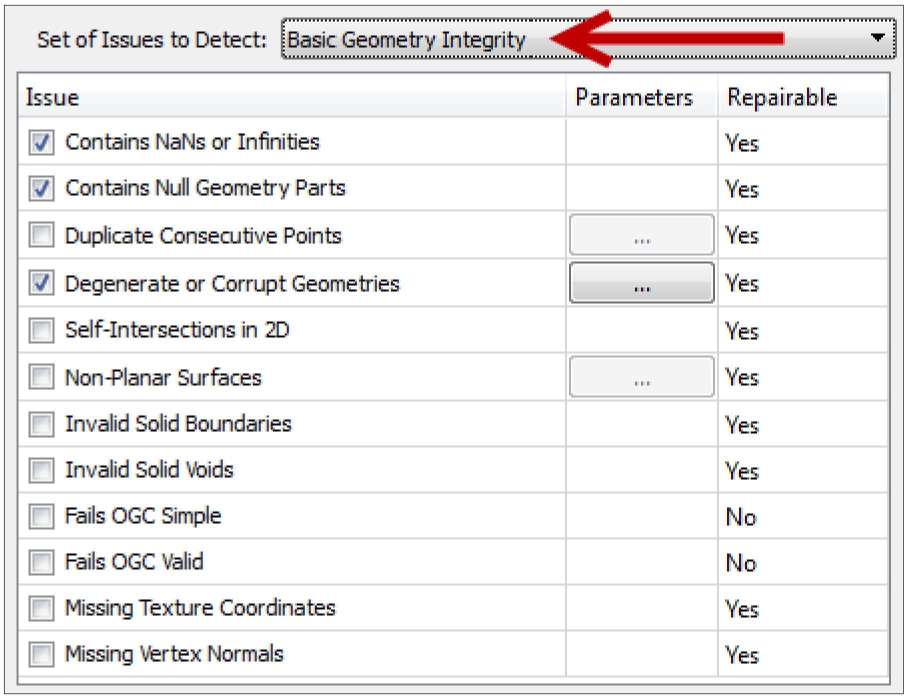
…or at the Feature Type level:


Spatial Type: GEOMETRY
Spatial Column Name
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):


Set of Issues to Detect: Basic Geometry Integrity

Issue	Parameters	Repairable
✓ Contains NaNs or Infinities		Yes
✓ Contains Null Geometry Parts		Yes
☐ Duplicate Consecutive Points	...	Yes
✓ Degenerate or Corrupt Geometries	...	Yes
☐ Self-Intersections in 2D		Yes
☐ Non-Planar Surfaces	...	Yes
☐ Invalid Solid Boundaries		Yes
☐ Invalid Solid Voids		Yes
☐ Fails OGC Simple		No
☐ Fails OGC Valid		No
☐ Missing Texture Coordinates		Yes
☐ 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.

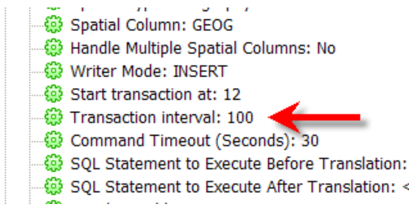
Transaction Interval
Transaction Interval is a SQL Server writer parameter that specifies the number of individual features to be written to the database before a commit action takes place.

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.



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: <

Bulk Insert
A further SQL Server writer parameter is Bulk Insert.

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.

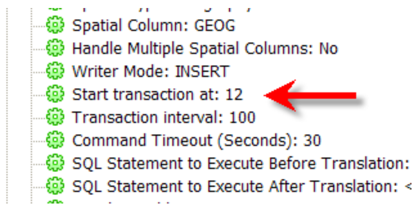
As a previous example showed, the only writer mode compatible with Bulk Insert is INSERT. Bulk Insert is not possible when the writer mode is set to either UPDATE or DELETE.

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

Safe Software Training Presentation
FME and Raster

FME 2008
Version

www.safe.com

SAFE SOFTWARE

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