Article

← FME Desktop (/S/Topic/0TO4Q000000QL9uWAG/Fme-...

Introduction to Stream Processing in FME

Jul 29, 2022 • Knowledge

Product Type
FME Desktop

FME Version
2022.0

Introduction

FME's stream processing capabilities allow you to leverage unbounded stream datasets without writing a single line of code. This tutorial will guide you through three different stream processing scenarios. In each scenario, we will walk through how to build the FME workflows and how to deploy the stream workspace to FME Server or FME Cloud.

Each scenario will highlight particular functionality in FME when it comes to authoring stream processing workspaces.

Content Overview

Part 4: Triggering an Event with the WindowChanged Output

Downloads

Starting and completed workspaces and data are available to download here (https://safe.my.salesforce.com/sfc/p/30000000ePES/a/4Q000000V79y/9NTbKzAwUYf7FZ.fLxhOTwCavyDjQVO5qzkiW9bFXsw).

Requirements

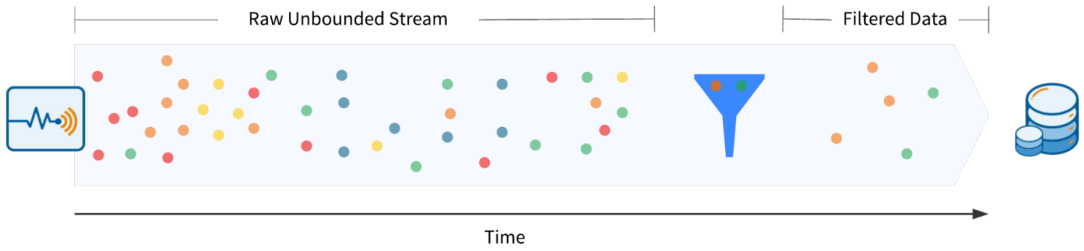
Unfortunately, we cannot provide public access to our Kafka queue due to the cost of running the stream at all times. Alternatively, replace the Creator and KafkaConnector in the provided workspaces with the contents of *MimicStream.fmwt* to imitate a stream data source.

Step-by-Step Instructions

Part 1: How to Work with High-Volume Streams in FME

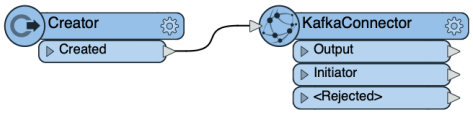
In this scenario, we're using a simulated vehicle fleet. Each vehicle reports their locations to a Kafka message queue every few seconds. There are a variety of vehicle types like sedans, SUVs and trucks, but we're only interested in tracking the trucks. Storing records from every vehicle would be very expensive and require far more storage space than needed. Instead, the records should be filtered so that we only store the truck data. We will also enhance the stream messages with spatial data.

This tutorial will show how to connect to a Kafka message queue, reduce data volumes in memory by filtering, and enhance incoming stream data into spatial data.



1. Connect to a Kafka Message Broker in Batch Mode

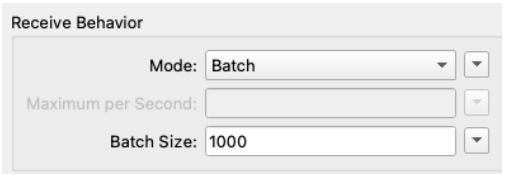
First, add a Creator transformer, and connect a KafkaConnector to the Creator. The single feature released from the Creator will initiate the KafkaConnector to run.



Open the KafkaConnector's parameters and enter the following connection details:

- **Credential Source:** Embedded
- **User Name:** <username>
- **Password:** <password>
- **Broker Host Name:** pkc-4nym6.us (http://pkc-4nym6.us)-east-1.aws.confluent.cloud
- **Broker Port:** 9092
- **Topic:** cov-fleet-location
- **Consumer Group ID:** simulate-geofence

Confirm that the KafkaConnector's request action is "Receive" and the receive mode is set to "Batch". Set the batch size to 1000. The KafkaConnector will only receive 1000 messages from the queue.



2. Run the KafkaConnector

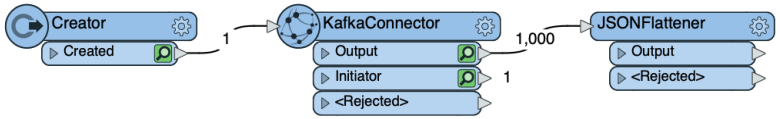
Run the workspace. The KafkaConnector will receive a batch of 1000 features from the message queue. When the workspace finishes running, you can view the feature cache at the output port to inspect the data.

	_creation_instance	_value	_timestamp	_topic	_connector_id
1	0	{"vehicle_id":"277715","vehicle_category":"light","vehicle_type":"sedan","ti...	1654812592....	cov-fleet-loc...	<null>
2	0	{"vehicle_id":"278994","vehicle_category":"light","vehicle_type":"trucks","...	1654812592....	cov-fleet-loc...	<null>
3	0	{"vehicle_id":"284753","vehicle_category":"light","vehicle_type":"trucks","...	1654812592....	cov-fleet-loc...	<null>
4	0	{"vehicle_id":"286047","vehicle_category":"light","vehicle_type":"trucks","...	1654812592....	cov-fleet-loc...	<null>
5	0	{"vehicle_id":"283541","vehicle_category":"light","vehicle_type":"trucks","t...	1654812592....	cov-fleet-loc...	<null>
6	0	{"vehicle_id":"278994","vehicle_category":"light","vehicle_type":"trucks","...	1654812592....	cov-fleet-loc...	<null>
7	0	{"vehicle_id":"285874","vehicle_category":"light","vehicle_type":"trucks","t...	1654812592....	cov-fleet-loc...	<null>
8	0	{"vehicle_id":"284753","vehicle_category":"light","vehicle_type":"trucks","...	1654812592....	cov-fleet-loc...	<null>
9	0	{"vehicle_id":"285874","vehicle_category":"light","vehicle_type":"trucks","t...	1654812592....	cov-fleet-loc...	<null>
10	0	{"vehicle_id":"278994","vehicle_category":"light","vehicle_type":"trucks","...	1654812592....	cov-fleet-loc...	<null>
11	0	{"vehicle_id":"279606","vehicle_category":"light","vehicle_type":"trucks","...	1654812592....	cov-fleet-loc...	<null>
12	0	{"vehicle_id":"286047","vehicle_category":"light","vehicle_type":"trucks","...	1654812592....	cov-fleet-loc...	<null>
13	0	{"vehicle_id":"281164","vehicle_category":"light","vehicle_type":"trucks","t...	1654812592....	cov-fleet-loc...	<null>
14	0	{"vehicle_id":"272532","vehicle_category":"light","vehicle_type":"suv","tim...	1654812602....	cov-fleet-loc...	<null>
15	0	{"vehicle_id":"275885","vehicle_category":"light","vehicle_type":"sedan","t...	1654812602....	cov-fleet-loc...	<null>
16	0	{"vehicle_id":"277724","vehicle_category":"light","vehicle_type":"sedan","t...	1654812602....	cov-fleet-loc...	<null>
17	0	{"vehicle_id":"270472","vehicle_category":"light","vehicle_type":"sedan","t...	1654812602....	cov-fleet-loc...	<null>

3. Extract the JSON Attributes

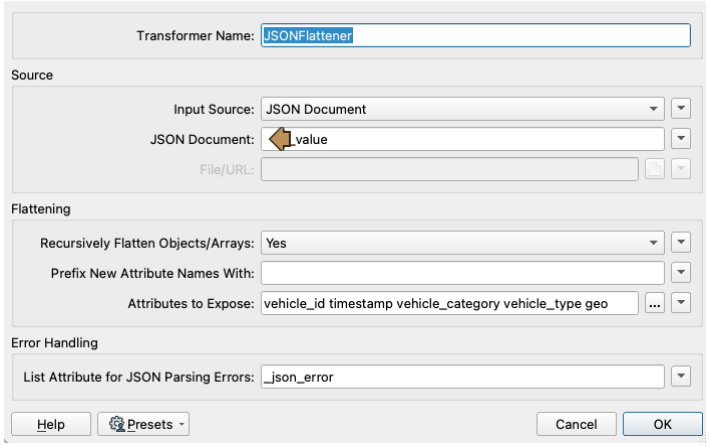
View the feature cache of the KafkaConnector output. Double-click one of the _value attribute values to view the JSON-formatted message contents. The JSON payload includes the vehicle type, and we'll want to use this information to filter the data.

First, we must extract the JSON attributes from the payload. Add a JSONFlattener after the KafkaConnector, and connect the Output port to the JSONFlattener's input.



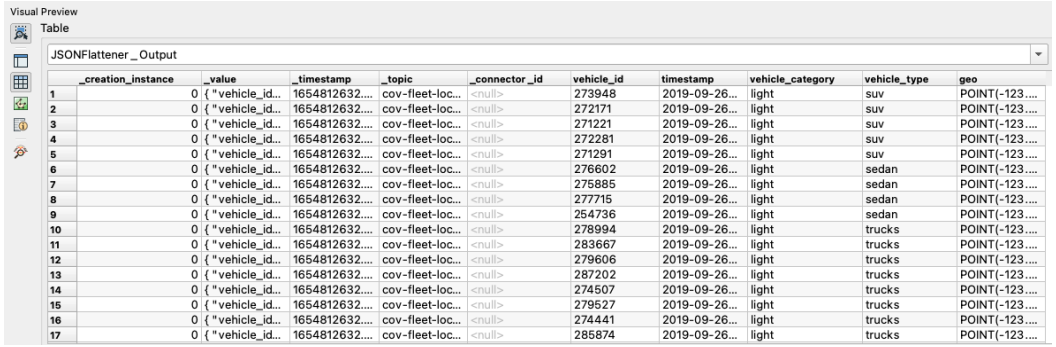
Open the JSONFlattener parameters. For the JSON Document parameter, select the dropdown menu > Attribute Value > _value. For the Attributes To Expose, click the ellipses button (...), and enter the following attributes in the pop-up window:

- vehicle_id
- timestamp
- vehicle_category
- vehicle_type
- geo



Tip: If you're working with JSON that has many attributes, save a sample JSON payload as a .json file. Use the Import button within the ellipses pop-up to read the JSON file, and FME will parse it and load the selected attributes.

Once you've filled in the attributes, click OK and OK again to save the parameter settings. Run the workspace to the JSONFlattener and inspect the output feature cache. The JSON attributes have now been extracted into separate attributes on each feature.

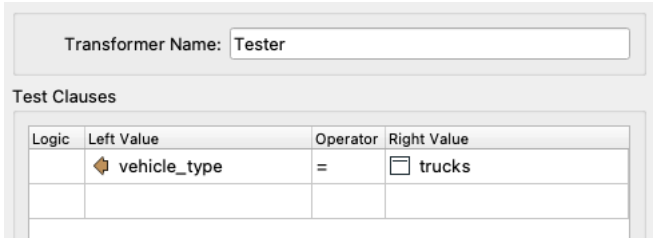


	_creation_instance	_value	_timestamp	_topic	_connector_id	vehicle_id	timestamp	vehicle_category	vehicle_type	geo
1	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	273948	2019-09-26...	light	suv	POINT(-123....
2	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	272171	2019-09-26...	light	suv	POINT(-123....
3	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	271221	2019-09-26...	light	suv	POINT(-123....
4	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	272281	2019-09-26...	light	suv	POINT(-123....
5	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	271291	2019-09-26...	light	suv	POINT(-123....
6	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	276602	2019-09-26...	light	sedan	POINT(-123....
7	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	275885	2019-09-26...	light	sedan	POINT(-123....
8	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	277715	2019-09-26...	light	sedan	POINT(-123....
9	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	254736	2019-09-26...	light	sedan	POINT(-123....
10	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	278994	2019-09-26...	light	trucks	POINT(-123....
11	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	283667	2019-09-26...	light	trucks	POINT(-123....
12	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	279606	2019-09-26...	light	trucks	POINT(-123....
13	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	287202	2019-09-26...	light	trucks	POINT(-123....
14	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	274507	2019-09-26...	light	trucks	POINT(-123....
15	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	279527	2019-09-26...	light	trucks	POINT(-123....
16	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	274441	2019-09-26...	light	trucks	POINT(-123....
17	0	{ "vehicle_id...	1654812632....	cov-fleet-loc...	<null>	285874	2019-09-26...	light	trucks	POINT(-123....

4. Filter the Features with a Tester

Now that we've extracted the vehicle_type attribute along with other useful information about each message, we can filter the data to just trucks. To do this, connect a Tester to the JSONFlattener output.

In the Tester parameters, set a clause to check if the vehicle_type = trucks. Click OK.



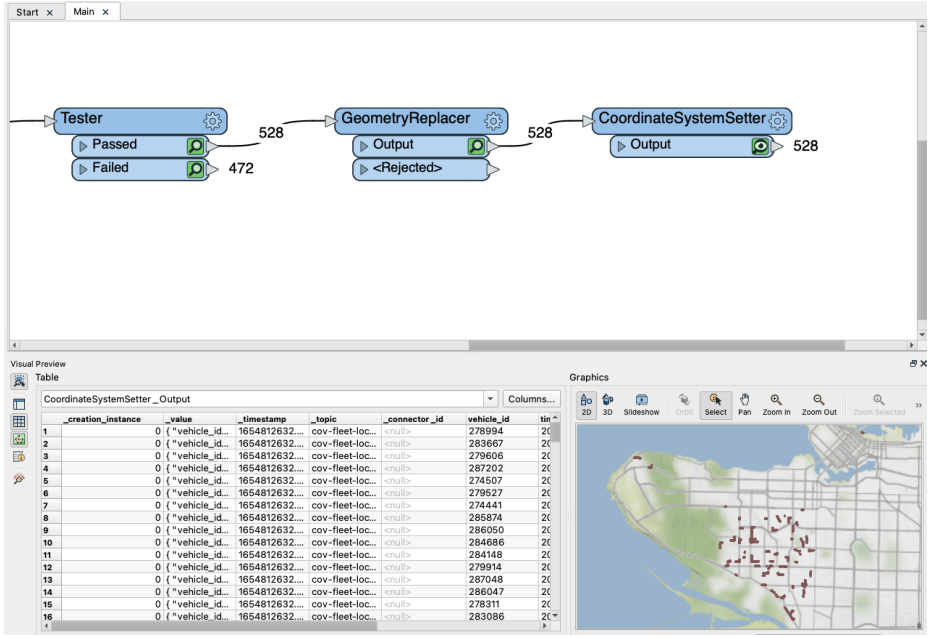
Run the workspace to the Tester and watch how only truck features are routed through the Passed output port.

5. Enhance the Features with Spatial Data

Stream data is usually non-spatial, but spatial-related attributes may be included in the data such as latitude/longitude coordinates to use in spatial analysis.

First, we need to generate points for each of our vehicles. Add a GeometryReplacer, and set the encoding to OGC Well Known Text and Geometry Source to the geo attribute. Leave the rest of the parameters as the default settings.

Our features don't have an associated coordinate system, so we must provide that information. Add a CoordinateSystemSetter, and select LL84. When running the workspace up to this transformer, you should see the points appear in Vancouver in the Visual Preview.

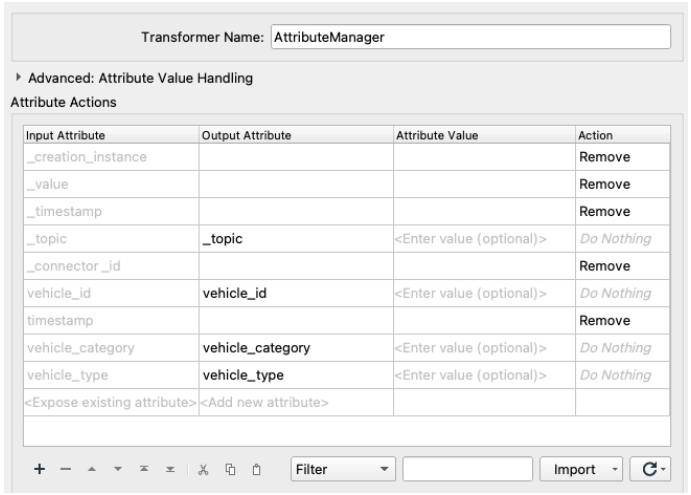


6. Clean up the Attributes Before Writing

We're just about ready to write the results, but there are several attributes that we don't care to store in our output destination such as the original JSON payload.

Add an AttributeManager and connect it to the CoordinateSystemSetter. Remove the five following attributes and optionally reorder the remaining attributes:

- _creation_instance, _value, _topic, _connector_id, timestamp



7. Write the Results to PostGIS

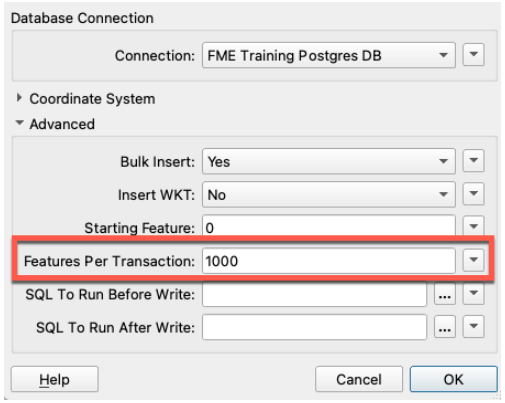
We've gone through the process of filtering and enhancing stream data, and now we're ready to write the features to a database.

Create a PostGIS writer and add a database connection to the following training PostGIS database:

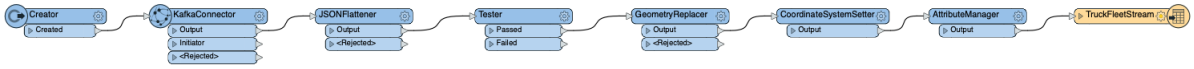
- **Host**: postgis.train.safe.com (http://postgis.train.safe.com)
- **Port**: 5432
- **Database**: fmedata
- **Username**: fmedata
- **Password**: fmedata

For the table definition, select Automatic. Once the writer feature type is added to the canvas, connect it to the AttributeManager. Set the table name to <unique>. Confirm the Feature Operation is "Insert" and the Table Handling is "Create If Needed". Click OK.

Select the writer feature type on the canvas, and click "Edit Parameters". Expand the Advanced section, and note that the Features Per Transaction is 1000 by default. The writer will commit a bulk insert of 1000 records at a time. This value can be adjusted depending on how frequently you want to write data to your database depending on the stream volumes. More information on this can be found in our article for Writing to Databases When Running in Stream Mode.



Your stream processing workspace is now writing filtered and spatially enhanced data to a PostGIS database! The workspace is ready to be published and run continuously on FME Server.



Part 2: Run the Workspace Continuously on FME Server

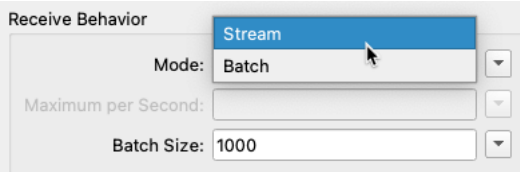
Now that we've gone through the process of building the workspace, we'll need to deploy it to FME Server to run continuously. To accomplish this, we will configure the KafkaConnector to run in Stream mode.

Why can't we just run the stream workspace in FME Workbench? It's possible to run the workspace in stream mode within FME Workbench to verify that it functions properly, but this method wouldn't be sufficient for a production environment. If FME Workbench closes, the workspace will stop running.

With FME Server Streams, you can run a stream processing workspace for an indefinite amount of time. In this tutorial, we'll demonstrate how to do this.

1. Set the KafkaConnector to Run in Stream Mode

We now want to publish the stream workspace to FME Server, and make sure the connector is set to Stream mode to run continuously. Open the KafkaConnector and set the receive behavior mode to Stream.



Note: Certain stream connectors like the WebSocketReceiver and JMSReceiver always run continuously, so you don't need to set the mode.

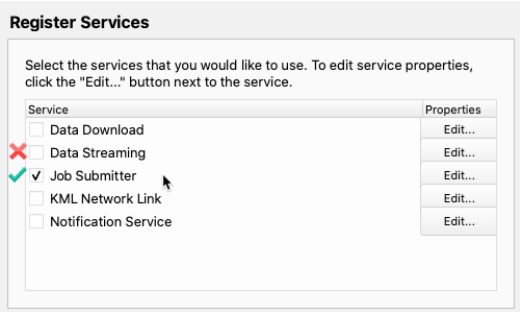
2. Publish the Workspace to FME Server

In the FME Workbench toolbar or the File menu, click Publish to FME Server. Select your existing FME Server web connection or create a new one by entering the server URL and your credentials. Click Continue.

Select the repository you'd like to publish the workspace to or create a new one. Click Continue. When prompted to upload the database connection to PostGIS, select the connection and click Continue.

In the next step, you'll need to select the Apache Kafka Connector package to upload it to FME Server. Many connectors are shipped as FME packages available through the FME Hub, and these need to be uploaded to FME Server through the publisher. Make sure the Kafka package is selected, and click Continue.

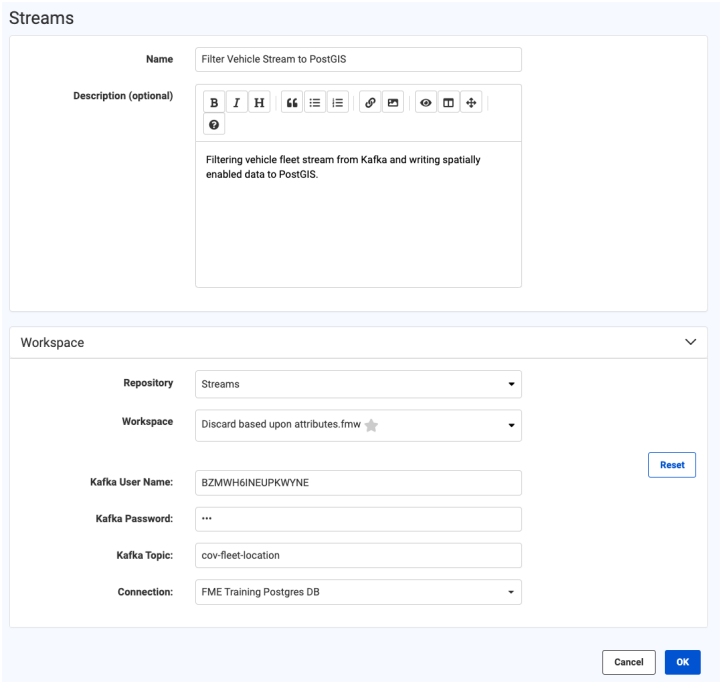
When registering the stream workspace to a service, only select the Job Submitter. Do not select Data Streaming because it's not related to stream processing. More details on the Data Streaming service can be found here. Click Publish.



3. Create an FME Server Stream

Log in to your FME Server. In the side menu, go to Streams > Build Stream. Enter a name for the stream, for example: Filter Vehicle Stream to PostGIS. Optionally add a description.

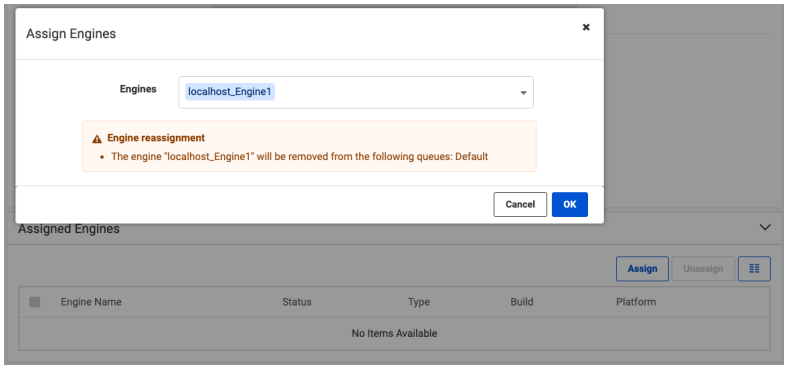
Next, select the workspace that you just published. No published parameters need to be changed. Click OK.



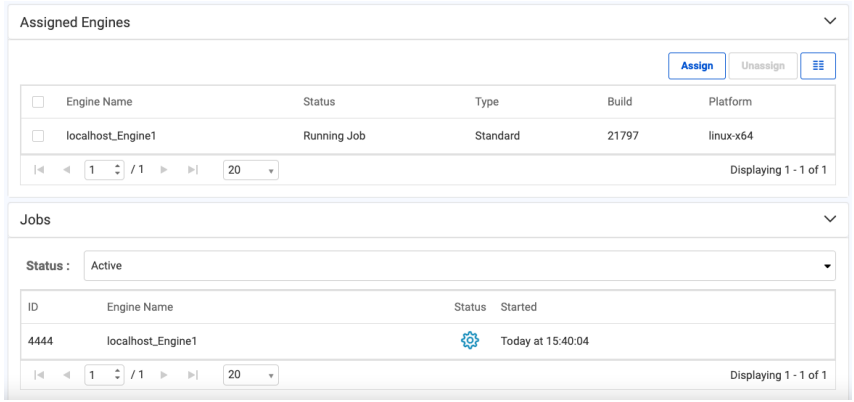
4. Assign an Engine and Run the Stream Workspace

The next page is where you can manage your stream. Streams are started automatically when creating them, but they will not run the stream workspace job until at least one FME Server Engine has been assigned to the stream so that it can run continuously.

Under the Assigned Engines section, click Assign. In the Engines dropdown menu, select one of your FME Server Engines. Note how the Engine will be removed from the queue it's currently assigned to. Click OK.



Within a few seconds, the stream job will start running and you can see it under the Jobs section. Clicking the job will open the translation log.

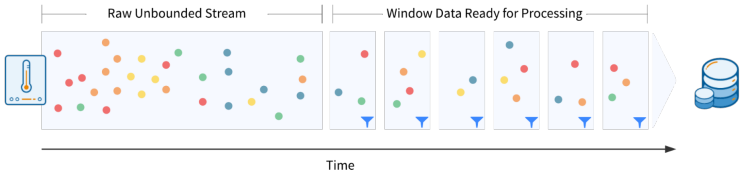


To stop the stream workspace, go back to the stream page and click the Stop button at the top. You've successfully created and deployed a stream processing workspace on FME Server!

Part 3: Time Windowing & Group By

When performing traditional batch processing in FME from a non-streaming data source (e.g. database or file), the data is finite with a defined beginning and end. You can read all of the data at once, and perform group-based analyses such as sorting and aggregating. Grouping is often based on location or attributes.

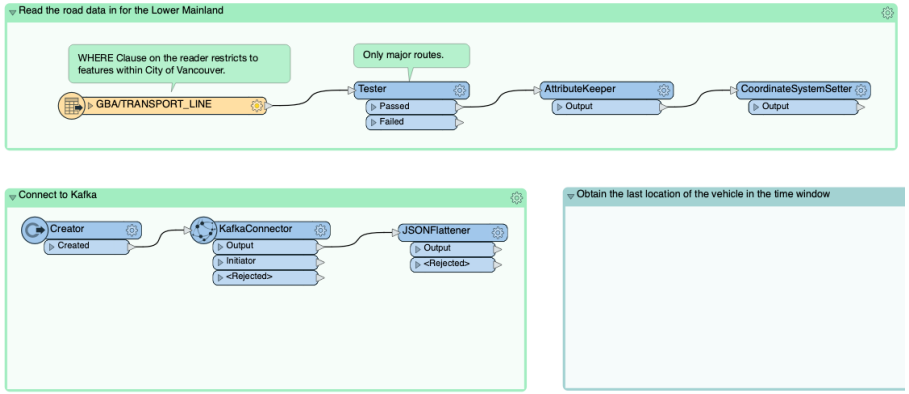
However, when reading a data stream into FME, the data is infinite and it will never finish loading. With this new data model, how can you perform group-based processing? This is where windowing comes into play. Windowing is the process of using time to break up a data stream into groups for filtering and analysis.



In this tutorial, we will use the TimeWindower transformer to separate the vehicle fleet stream data into groups based on 30-second windows. If we were to use a longer time window, we'd be reducing the data volume even further. Data in each window can then be thinned and snapped to a road network dataset.

1. Review the Workspace - Reading the Data

We are reading in the stream GPS data via the KafkaConnector and the road network data via a File Geodatabase reader.



Using a Tester, the road network data is filtered as we are only interested in snapping vehicles to the major routes. Attributes are tidied up using the AttributeKeeper, and the data is then assigned a coordinate system using the CoordinateSystemSetter.

For the GPS data, the configuration of the KafkaConnector and JSONFlattener are identical to the settings defined in part 1. Note, by default the KafkaConnector is in Stream mode.

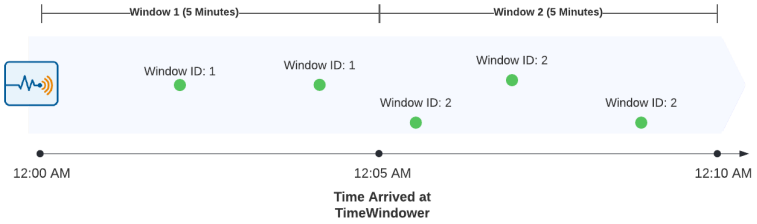
2. Window the Stream with a TimeWindower

We will now window the GPS data stream to reduce data volumes. Each vehicle reports its location every 5 seconds but we only need an accuracy of one location per vehicle per 30 seconds.

Add a TimeWindower after the JSONFlattener and enter the following parameters:

- **Window Duration**: 30
- **Time Units**: Seconds
- **Time Window Starts**: With First Feature
- **Data Has Timestamps**: No
- **Window ID Attribute**: _window_id
- **ID Type**: Window Number

Based on when the feature arrives at the TimeWindower, a different window ID will be assigned to the features for each 30-second period. That Window ID can then be used downstream to group the data for processing. The diagram below shows how window IDs are assigned to features when the window duration is 5 minutes.



3. Thinning the GPS Data

Using the Sorter and Sampler transformers, the GPS data can now be thinned by using the Window ID as a Group By. When sorting each vehicle in chronological order, the Sampler can extract the last feature of each vehicle which is the most recent location.

Add a Sorter transformer, enable Group Processing and set the following:

- **Group By**: _window_id
- **Complete Groups**: When Group Changes (Advanced)
- **Sort by**:
  - path_id, Alphabetical, Ascending
  - timestamp, Alphabetical, Ascending

Now, the GPS points for each vehicle will be grouped together in the window with the most recent location at the end.

Tip: This process may be challenging to visualize. To see how it works, run the KafkaConnector in Batch mode with ~5000 features and inspect the feature caches in the Sorter and Sampler.

The Sampler transformer can be used to get the last location for each vehicle. Enable Group Processing and set the following:

- **Group By**: _window_id, path_id
- **Complete Groups**: When Group Changes (Advanced)
- **Sampling rate (N)**: 1
- **Sampling Type**: Last N Features
- **Randomize Sampling**: No

4. Snap the Stream Data to the Road Network

We have now thinned the data, but some of the points aren't lined up well with the road network due to minor inaccuracies with the GPS. To fix this, we'll snap the vehicle points to the road network.

Add a GeometryReplacer to extract the geometry defined in text in the JSON and create point geometries.

- **Geometry Encoding**: OGC Well Known Text
- **Geometry Source**: geometry
- **Remove Attributes**: Yes

Add a Reprojector to reproject to the same coordinate system as the roads

- **Source Coordinate System**: EPSG:4326
- **Destination Coordinate System**: NAD83.BC/Albers

Next, we need to add a NeighborFinder to snap the GPS points to the road network. Attach the road data to the Candidate port and the GPS data to the Base port. Then set the following:

- **Input**: Bases and Candidates
- **Candidates First**: Yes
- **Maximum Distance**: 5

- **Attribute Accumulation**: Merge Attributes
- **Accumulation Mode**: Merge Candidates
- **Conflict Resolution**: Use Base

Group By processing is not required here as we are simply assessing the GPS points individually against the road network.

Run the workspace, features should exit from the MatchedBase port of the NeighborFinder.

5. Switch to Window Based Upon Timestamp

At the moment, data is being windowed based on when it arrives at the TimeWindower transformer. However, the data has a timestamp on it which represents the actual event time. Let's change the workflow to window using the event time instead. The thing we have to account for here is that there is a lag between when the event actually happened and when the event arrives at the transformer. We will account for this by setting a tolerance.

Open the TimeWindower and set the following:

- **Data Has Timestamp**: Yes
- **Timestamp Attribute**: timestamp
- **Order Type**: Chronological Order
- **Tolerance Interval**: 10
- **Tolerance Time Units**: Seconds

This tolerance setting effectively leaves the window open for 10 extra seconds, this gives a chance for all data to arrive for processing before the window is closed. The features that arrive after the window is closed are output via the OutOfSequence port. To learn more about this, please see our article on Windowing Data Streams.

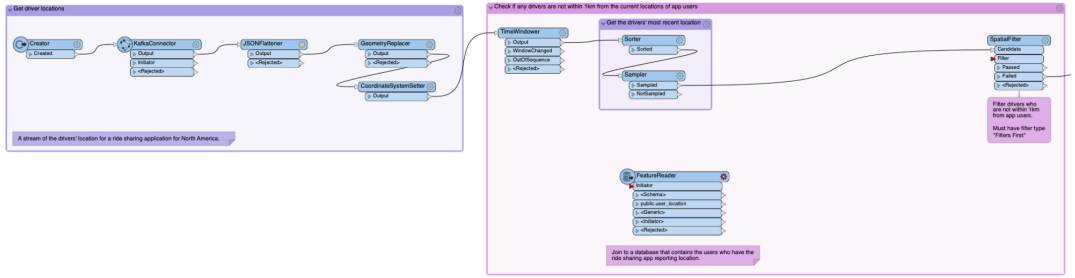
Part 4: Triggering an Event with the WindowChanged Output

In part 3 we joined the stream of data to another data source. In that example the dataset was static so reading it in only once when the workspace starts running is fine. However, if you need to join the stream to a dataset that is constantly changing, a different approach is required using the TimeWindower.

1. Review the Workspace

In this scenario, we are operating a ride-hailing app. We are reading in a stream of GPS points that represents the driver's locations, and we want updates on app user locations to see if our drivers are close enough to them.

The driver stream data is extracted and then converted into point geometry so we can perform a spatial operation on it downstream. Using the TimeWindower, the data is windowed into 5-minute windows.



2. Reconfigure the TimeWindower

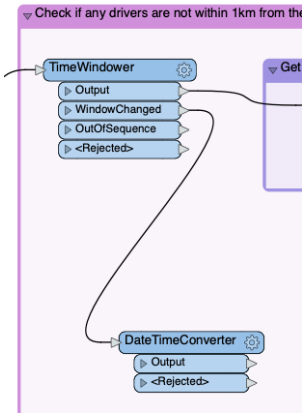
Once the driver GPS data has been thinned and we have the last location of each vehicle during the 5-minute window, we want to perform a spatial operation to identify the current location of app users who are not within 1km of drivers. The crucial part here is “current” location. We are in stream mode and need to join the Kafka stream to the users’ location. To do this, we use the WindowChanged port on the TimeWindower transformer.

Open the TimeWindower transformer and set the ID Type to be Window Start Time. This is important as we need a time to query the database.

3. Prepare Features for Database Query

The Window Start Time is in seconds since the Unix Epoch so we need to change this to be a format that PostgreSQL can accept. Add a DateTimeConverter, and connect it to the TimeWindower’s WindowChanged port. Set the following parameters:

- **Datetime Attributes**: _window_id
- **Input Format**: %s
- **Output Format**: %Y-%m-%d %H:%M:%S

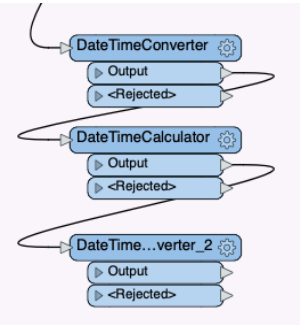


The query to PostgreSQL is a time range as we want to retrieve all of the users that have logged a location within the last 5 minutes. To do this, we need to pass in a start time and end time attributes to be used in the query. The _window_id is the start time; let’s calculate the end time. We need the end time to represent the end of the window; since the window is 5 minutes we need to add that on. Connect a DateTimeCalculator to the DateTimeConverter and set the following:

- **Datetime**: _window_id
- **Operator**: Add
- **Minutes**: 5
- **Result**: _end_window

Now we also need to format the _end_window date time to be the format %Y-%m-%d %H:%M:%S. Add another DateTimeConverter after the DateTimeConverter and use the following settings:

- **Datetime Attributes**: _end_window
- **Input Format**: Auto Detect
- **Output Format**: %Y-%m-%d %H:%M:%S



4. Retrieve User Location for the Current Window

Next, we need to query the PostgreSQL user_location table. Connect the FeatureReader to the DateTimeConverter_2 and look at the settings. The key thing to understand is the WHERE Clause.

```
timestamp >= '@Value(_window_id)'
AND timestamp < '@Value(_end_window)'
ORDER BY device_id
```

This will retrieve all users from the database that had an event time that fell within the current window.

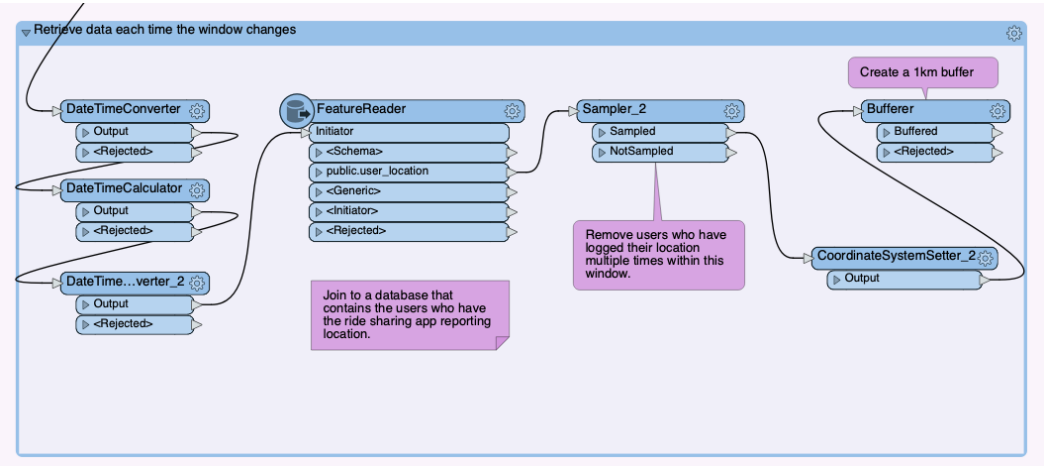
Since each user's location will have been logged multiple times within the window, we need to get the last known location for each user. The data returned from PostgreSQL is sorted by device_id so we can use a Sampler to get a single location for each user. Connect a Sampler to the public.user_location port.

- **Group By**: device_id
- **Complete Groups**: When Group Changes (Advanced)
- **Sampling Rate**: 1
- **Sampling Type**: Last N Features

5. Identify Drivers Who are Not Within 1km of Any User

Add a CoordinateSystemSetter after the Sampler_2 and set the Coordinate System to EPSG:4326. Next, let's create a 1km buffer around all of the users. Add a Bufferer transformer and set the following:

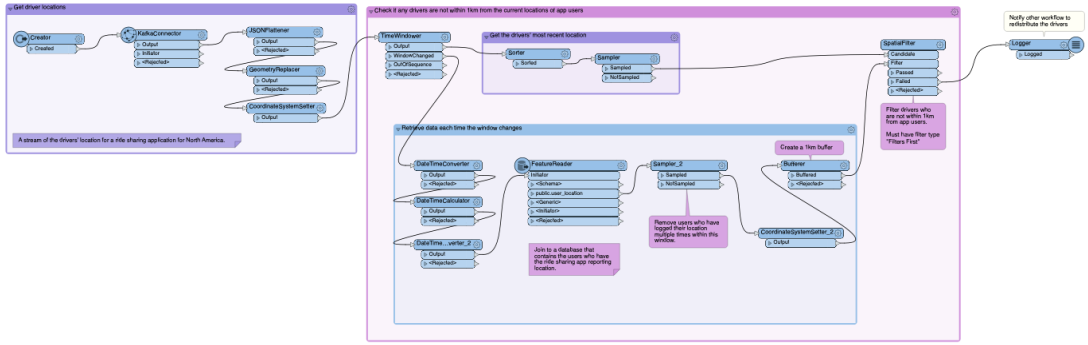
- **Buffer Type**: Area (2D)
- **Buffer Distance**: 1
- **Buffer Distance Unit**: Kilometers



Now we are ready to join the location of all of the app users and the driver's location so we can try to identify the drivers that are not within 1km of the users. Add a SpatialFilter, connect the stream of drivers to the Candidate port and the users to the Filter port. Set the following values:

- **Group Processing**: Enabled
- **Group By**: _window_id
- **Complete Groups**: When Group Changes (Advanced)
- **Filter Type**: Filters First
- **Spatial Predicates to Test**: "Filter OGC-Contains Candidate"

The features that exit the Failed port are all of the drivers who are not within 1km of any users.



Conclusion

You've now completed this tutorial series on getting started with stream processing in FME! To continue learning about different stream processing scenarios, authoring tips, considerations with databases, and more, be sure to check out our FME and Stream Processing landing page for more tutorials and articles.

Data Attribution

The data used here originates from data made available by the City of Vancouver, British Columbia (https://opendata.vancouver.ca/pages/licence/). It contains information licensed under the Open Government License - Vancouver.

Additional Resources

Desktop Tips for Working with Continuous Data Streams (/s/article/Desktop-Tips-for-Working-with-Continuous-Data-Streams)

Writing to Databases When Running in Stream Mode (/s/article/Writing-to-databases-when-running-in-stream-mode)

Windowing Data Streams in FME (/s/article/Windowing-Data-Streams-in-FME)

Filtering Unbounded Data Streams (/s/article/Filtering-Unbounded-Data)

Spatial Analysis on Unbounded Data Streams (/s/article/Proximity-Analysis-on-Unbounded-Data-Streams)

First Published Date
7/29/2022, 10:51 PM

Last Published Date
7/29/2022, 10:51 PM

Log In to Comment

Follow

Files (1) (/s/relatedlist/ka14Q000001DX20QAG/AttachedContentDocuments)

IntroToStreams
Aug 9, 2022 • 1.3MB • zip

View All
(/s/relatedlist/ka14Q000001DX20QAG/AttachedContentDocuments)

Related Articles

FME and Stream Processing (/s/article/Getting-Started-with-Stream-Processing-in-FME)

Introduction to MapTextLabeller (/s/article/introduction-to-maptextlabeller)

How To Use Parallel Processing in FME (/s/article/how-to-use-parallel-processing-in-fme)

Introduction to the MapnikRasterizer (/s/article/introduction-to-mapnikrasterizer)

Introduction to Scheduling and Change Detection Workflows in FME Server (/s/article/introduction-to-scheduling-and-change-detection-wo)

Land Acknowledgement —

Safe Software respectfully acknowledges that we live, learn and work on the traditional and unceded territories of the Kwantlen, Katzie, and Semiahmoo First Nations.