

PostGIS Reader/Writer

FORMAT NOTES:

This format is not supported by FME Base Edition.

Overview

PostGIS is a geometric layer over a PostgreSQL Object-Relational Database Management System (ORDBMS) that provides geometry and Spatial Reference System (SRS) handling. The PostGIS reader/writer module provides the Feature Manipulation Engine (FME) with the ability to read geometric PostGIS data as well as underlying attribute data stored in PostgreSQL.

The PostGIS reader/writer is specifically designed to handle the geometric and SRS portions of the data. When reading attribute-only tables from PostgreSQL, the PostgreSQL reader/writer should be used instead. The PostGIS reader/writer communicates directly with the PostgreSQL libpq interface for maximum throughput.

This chapter assumes familiarity with PostGIS and PostgreSQL, the attribute and geometry types supported, and its indexing mechanisms.

For more information, please see the PostgreSQL home at

<http://www.postgresql.org/>

and the PostGIS home at

<http://postgis.refractory.net/>

PostGIS Quick Facts

Format Type Identifier	PostGIS
Reader/Writer	Both
Licensing Level	Professional
Dependencies	None
Dataset Type	Database
Feature Type	Table name
Typical File Extensions	None
Automated Translation Support	Yes
User-Defined Attributes	Yes
Coordinate System Support	Yes
Generic Color Support	No
Spatial Index	Always
Schema Required	Yes
Transaction Support	Yes
Geometry Type	postgis_type

Geometry Support			
Geometry	Supported?	Geometry	Supported?
aggregate	yes	point	yes
circles	no	polygon	yes
circular arc	no	raster	no
donut polygon	yes	solid	no
elliptical arc	no	surface	no
ellipses	no	text	no
line	yes	z values	yes
none	yes		

Reader Overview

FME considers a PostGIS dataset to be a database containing a collection of relational tables together with their corresponding geometries. The tables must be defined in the mapping file before they can be read. Arbitrary `WHERE` clauses and joins are fully supported, as well as an entire arbitrary SQL `SELECT` statement; however, the user then assumes responsibility for the correctness of the statement or clause including quoting where necessary. Support for `@SQL` and `@Relate` functions has also been added.

When reading from the PostGIS/PostgreSQL database, each table is considered a feature type in FME and each row of a table at least one feature in FME. In the case of heterogeneous geometry collections, they may become more than one FME feature.

The basic reading process involves opening a connection to the database, querying metadata, and querying data. The data is read using a text cursor and rows are fetched to the client machine in batches of 10000 by default. There is one cursor per input table.

If NULL geometries are read, they are treated as non-geometry features and the attributes are preserved.

Table and column names are truncated at 64 characters. If duplicate names are produced by truncation, the behavior is undetermined. Please ensure that table names comply with PostgreSQL naming conventions.

Spaces and special characters are permissible in both table and column names. Case sensitivity has also been implemented, so table and column names are no longer changed to lowercase.

Table listing support when using the PostGIS settings boxes has been improved to avoid errors with schemas and tables that do not exist, or are inconsistent with the PostGIS metadata.

UNICODE support has been added to work with a client's system encoding. Although there is no way to explicitly specify the encoding, the client is assumed to have entered data and created tables and columns in the encoding of their operating system. Multiple system encodings are now supported via the native PostgreSQL conversions between client and server, particularly if the server encoding is set to UNICODE.

Older schema directives have been removed and qualified table naming is now supported in the form `<schemaname>.<tablename>`. Additionally, the schema search path is now read and interpreted to determine a user's default schema when writing and the available schema to read from when reading. Failing a valid schema search path, the default public schema will be used for newer databases.

Reader Directives

The directives that are processed by the PostGIS reader are listed below. The suffixes shown are prefixed by the current `<ReaderKeyword>_` in a mapping file. By default, the `<ReaderKeyword>` for the PostGIS reader is `POSTGIS_IN`.

DATASET/DATABASE

Required/Optional: *Required*

This specifies the name of the PostGIS-enabled PostgreSQL database. The database must exist in the ORDBMS.

```
POSTGIS_DATASET testdb
```

Workbench Parameter: [<WorkbenchParameter>](#)

HOST

Required/Optional: *Required*

This specifies the machine running the PostGIS/PostgreSQL ORDBMS as either an IP address or host name. The database must have proper permissions and be set up to accept TCP/IP connections if connecting from a remote machine.

```
POSTGIS_IN_HOST myserver
```

[Workbench Parameter: <WorkbenchParameter>](#)

PORT

Required/Optional: *Required*

When connecting remotely, this specifies the TCP/IP port on which to connect to the ORDBMS service. The default port is 5432.

```
POSTGIS_IN_PORT 5432
```

[Workbench Parameter: <WorkbenchParameter>](#)

USER_NAME

Required/Optional: *Required*

The name of user who will access the database. The named user must exist with appropriate PostgreSQL permissions. The default user name is `postgres`.

```
POSTGIS_IN_USER_NAME postgres
```

[Workbench Parameter: <WorkbenchParameter>](#)

PASSWORD

Required/Optional: *Optional*

The password of the user accessing the database. This parameter is optional when using a trusted connection. Other authentication types such as password, or MD5 require this parameter to be set.

```
POSTGIS_IN_PASSWORD secret
```

[Workbench Parameter: <WorkbenchParameter>](#)

DEF

Required/Optional: *Required*

The syntax of the definition is:

```
POSTGIS_DEF <tableName>                                     \  
  [postgis_where_clause      <whereClause>]               \  
  [<fieldName> <fieldType>] +
```

OR

```
POSTGIS_DEF <queryName>                                     \  
  [postgis_sql_statement      <sqlQuery>]
```

The `<tableName>` must match a PostGIS table in the database. This will be used as the feature type of all the features read from the table. The exception to this rule is when using the `sql_statement` directive. In this case, the `DEF` name may be any valid alphabetic identifier; it does not have to be an existing table name – rather, it is an identifier for the custom SQL query. The feature type of all the features returned from the SQL query are given the query name.

The `<fieldType>` of each field must be given, but it is not verified against the database definition for the field. In effect, it is ignored.

The exception to this is the `geometry` field type which is not placed in the `DEF`. This is stored separately in the `geometry_columns` table of the PostgreSQL database which maps geometry information to the database and table name.

The definition allows specification of separate search parameters for each table. If any of the configuration parameters are given, they will override, for that table, whatever global values have been specified by the reader directives listed above. If any of these parameters is not specified, the global values will be used.

The following table summarizes the definition line configuration parameters:

Parameter	Contents
<code>where_clause</code>	This specifies the SQL WHERE clause applied to the attributes of the layer's features to limit the set of features returned. If this is not specified, then all the tuples are returned. This directive will be ignored if the <code>sql_statement</code> is present.
<code>sql_statement</code>	This specifies an SQL SELECT query to be used as the source for the results. If this is specified, the PostGIS reader will execute the query, and use the resulting rows as the features instead of reading from the table <code><queryName></code> . All returned features will have a feature type of <code><queryName></code> , and attributes for all columns selected by the query. All parameters that specify a spatial constraint are ignored if an <code>sql_statement</code> is supplied. If selecting a geometry column from a PostGIS table, avoid the use of geometry column format modifiers such as <code>AsBinary()</code> , <code>AsText()</code> , <code>AsWKT()</code> , or <code>ASWKB()</code> since this obscures the fact that we have a geometry column and not just some text or byte attribute column.

If no `<whereClause>` is specified, all rows in the table will be read and returned as individual features. If a `<whereClause>` is specified, only those rows that are selected by the clause will be read. Note that the `<whereClause>` does not include the word "where".

The PostGIS reader allows one to use the `sql_statement` parameter to specify an arbitrary SQL SELECT query on the `DEF` line. If this is specified, FME will execute the query, and use each row of data returned from the query to define a feature. Each of these features will be given the feature type named in the `DEF` line, and will contain attributes for every column returned by the SELECT. In this case, all `DEF` line parameters regarding a WHERE clause or spatial querying are ignored, as it is possible to embed this information directly in the text of the `<sqlQuery>`.

The following example selects rows from the table `ROADS`, placing the resulting data into FME features with a feature type of `MYROADS`. Imagine that `ROADS` defines the geometry for the roads, and has a numeric field named `ID`, a text field named `NAME` and a geometry column named `GEOM`.

```
POSTGIS_DEF MYROADS
```

```
sql_statement`SELECT id, name, geom FROM ROADS`
```

Workbench Parameter: [<WorkbenchParameter>](#)

IDs

Required/Optional: *Optional*

This optional specification is used to limit the available and defined database tables files that will be read. If no IDs are specified, then no tables are read. The syntax of the IDs directive is:

```
POSTGIS_IDS <featureType1>          \  
  <featureType2>                    \  
  <featureTypeN>
```

The feature types must match those used in DEF lines.

The example below selects only the ROADS table for input during a translation:

```
POSTGIS_IDS ROADS
```

Workbench Parameter: [<WorkbenchParameter>](#)

MINX, MINY, MAXX, MAXY

Required/Optional: *Optional*

These directives when used together specify the spatial extent of the feature retrieval. Only features that interact with the bounding box defined by these directive values are returned. If this is not supplied, all features will be returned. If either min value is greater than the corresponding max value, the values will be swapped. If less than the entire set of four values are supplied, the supplied values will be ignored and all features will be returned.

The syntax of the directives is:

```
POSTGIS_IN_MINX <minX>  
POSTGIS_IN_MINY <minY>  
POSTGIS_IN_MAXX <maxX>  
POSTGIS_IN_MAXY <maxY>
```

The example below selects a small area for extraction:

```
POSTGIS_IN_MINX 25.6  
POSTGIS_IN_MINY 59.0  
POSTGIS_IN_MAXX 79.2  
POSTGIS_IN_MAXY 124.5
```

Workbench Parameter: [<WorkbenchParameter>](#)

SEARCH_ENVELOPE

Required/Optional: *Optional*

This directive is used to specify the spatial extent of the feature retrieval. Only features that interact with the bounding box defined by these directive values are returned. If this is not supplied, all features will be returned. If either min value is greater than the

corresponding `max` value, the values will be swapped. If less than the entire set of four values are supplied, the supplied values will be ignored and all features will be returned.

This directive supersedes the use of the `POSTGIS_IN_MINX`, `POSTGIS_IN_MINY`, `POSTGIS_IN_MAXX`, `POSTGIS_IN_MAXY` directives which are only used as a secondary fallback when this directive is empty.

The syntax of the **`POSTGIS_IN_SEARCH_ENVELOPE`** directive is:

```
POSTGIS_IN_SEARCH_ENVELOPE <minX> <minY> <maxX> <maxY>
```

The example below selects a small area for extraction:

```
POSTGIS_IN_SEARCH_ENVELOPE 25.6 59.0 79.2 124.5
```

[Workbench Parameter: <WorkbenchParameter>](#)

SEARCH_ENVELOPE_COORDINATE_SYSTEM

Required/Optional: *Optional*

This directive specifies the coordinate system of the search envelope if it is different than the coordinate system of the data. The `COORDINATE_SYSTEM` directive, which specifies the coordinate system associated with the data to be read, must always be set if the `SEARCH_ENVELOPE_COORDINATE_SYSTEM` directive is set.

If this directive is set, the minimum and maximum points of the search envelope are reprojected from the `SEARCH_ENVELOPE_COORDINATE_SYSTEM` to the reader `COORDINATE_SYSTEM` prior to applying the envelope.

The syntax of the `SEARCH_ENVELOPE_COORDINATE_SYSTEM` directive is:

```
<ReaderKeyword>_SEARCH_ENVELOPE_COORDINATE_SYSTEM <coordinate system>
```

[Workbench Parameter: Search Envelope Coordinate System](#)

SEARCH_METHOD

Required/Optional: *Optional*

This directive is used to specify the spatial relationship between the provided bounding box and the geometries in the geometry column of the table. There are two types of operation: Maximum Bounding Rectangle (MBR) operations, and full spatial operations. MBR operations will determine adherence to a given spatial relationship using only the bounding box of the geometry, whereas full spatial relationships will use the actual geometry itself.

Full spatial relationship comparisons are only available if GEOS is enabled on the PostGIS server. If not, all envelope comparisons will be made using the default MBR operation `MBR_OVERLAPS`.

The syntax of the **`POSTGIS_IN_SEARCH_METHOD`** directive is:

```
POSTGIS_IN_SEARCH_METHOD <spatial_relationship>
```

[Workbench Parameter: <WorkbenchParameter>](#)

FEATURES_PER_FETCH

Required/Optional: *Optional*

In order to avoid loading all the features in memory at once when reading a large dataset, cursors are used to retrieve the rows from the database. This optional directive specifies the number of rows to be read at one time from the cursor for a given query. The default is 10000 rows and should be sufficient in most cases. However this may need to be lowered or raised depending on the capabilities of the specific hardware in use and the data being read.

The example below selects a small set of features per extraction:

```
POSTGIS_IN_FEATURES_PER_FETCH 5000
```

[Workbench Parameter: <WorkbenchParameter>](#)

SIMPLIFY_AGGREGATES

Required/Optional: *Optional*

Specifies that we should split multipoints, multilinestrings and multipolygons that contain only one element into their corresponding simple element type, i.e., point, linestring or polygon. The default is `YES` for everything but PostGIS-to-PostGIS translations. When the destination format is also PostGIS, this gets automatically set to `NO` at mapping file or workspace generation time to preserve the values of the geometries.

[Workbench Parameter: <WorkbenchParameter>](#)

RETRIEVE_ALL_SCHEMAS

Required/Optional: *Optional*

This specification is only applicable when generating a mapping file, generating a workspace or when retrieving schemas in a FME Objects application.

This optional specification is used to tell the reader to retrieve the names and the schemas of all the tables in the source database. If this value is not specified, then it is assumed to be `No`. When set to `Yes`, indicates to the reader to return all the schemas of the tables in the database.

The syntax of the `RETRIEVE_ALL_SCHEMAS` directive is:

```
POSTGIS_RETRIEVE_ALL_SCHEMAS Yes
```

[Workbench Parameter: <WorkbenchParameter>](#)

RETRIEVE_ALL_TABLE_NAMES

Required/Optional: *Optional*

This specification is only applicable when generating a mapping file, generating a workspace or when retrieving schemas in a FME Objects application.

Similar to `RETRIEVE_ALL_SCHEMAS`: this optional specification is used to tell the reader to only retrieve the table names of all the tables in the source database. If

[<fieldName> <fieldType>][,<indexType>]*

The table definition allows control of the table that will be created. If the table already exists, the majority of the `postgis_` parameters will be ignored and need not be given. If the fields and types are listed, they must match those in the database.

If the table does not exist, then the field names and types are used to first create the table. In any case, if a `<fieldType>` is given, it may be any field type supported by the target database.

The configuration parameters present on the definition line are described in the following table:

Parameter	Contents
<code>tableName</code>	The name of the table to be written. If a table with the specified name exists, it will be overwritten if either the <code>postgis_overwrite_table DEF</code> line parameter set to <code>YES</code> or if the global writer directive type <code>postgis_out_overwrite</code> is set to <code>YES</code> . Otherwise the table will be appended. Valid values for table names include any character string devoid of SQL-offensive characters and less than 32 characters in length.
<code>postgis_type</code>	The type of geometric entity stored within the feature. The valid values for the type are listed below: <code>postgis_point</code> <code>postgis_line</code> <code>postgis_area</code> <code>postgis_geometrycollection</code> <code>postgis_none</code>
<code>postgis_mode</code>	The the default operation mode of the feature type in terms of the types of SQL statements sent to the database. Valid values are <code>INSERT</code> , <code>UPDATE</code> , <code>DELETE</code> and <code>INHERIT_FROM_WRITER</code> . Note that <code>INSERT</code> mode allows for only <code>INSERT</code> operations where as <code>UPDATE</code> and <code>DELETE</code> can be overwritten at the feature levels. <code>INHERIT_FROM_WRITER</code> simply indicates to take this value from the writer level and not to override it at the feature type level. Default: <code>INHERIT_FROM_WRITER</code>
<code>postgis_geometry_column</code>	This specifies the name of the column to be created that will hold the geometry when creating a new PostGIS table. Default: <code>geom</code>
<code>postgis_srid</code>	This specifies the spatial referencing information for the geometry in the table. By default, this value is <code>INHERIT_FROM_WRITER</code> which uses the conversion of the FME coordinate system of the writer into an SRID as the SRID for the given table. Alternatively, a specific integer SRID value may be specified. Specified SRID values should correspond to an existing the spatial reference identifier value stored in the (SRID) column in the global table <code>spatial_ref_sys</code> . All geometry within a given table must have the same spatial referencing. If <code>postgis_srid</code> is not specified, tables will be created with the SRID of the writer coordinate system. If empty SRIDs are desired, the value for the SRID field can be set to -1 indicating no spatial reference system.

Parameter	Contents
<code>postgis_drop_table</code>	This specifies that if the table exists by this name, it should be dropped and recreated before any features are written to it. This parameter, along with <code>postgis_truncate_table</code> , deprecates the older <code>postgis_overwrite_table</code> parameter.
<code>postgis_truncate_table</code>	This specifies that if the table exists by this name, it should be truncated before any features are written to it. This parameter, along with <code>postgis_drop_table</code> , deprecates the older <code>postgis_overwrite_table</code> parameter.
<code>postgis_create_with_ids</code>	Create the table including a system OID column as a unique identifier. If set to no, then the OID column is not created. Default: yes
<code>postgis_create_with_geometry_index</code>	Create a GiST index on the geometry column of the table (as long as one exists). The indexing of the geometry column is required for spatial query performance. Default: yes
<code>postgis_vacuum_analyze</code>	Perform the database function to vacuum and analyze the table once successfully written. This will build statistics for the table. Default: yes
<code>postgis_multi_geometry</code>	This specifies whether the PostGIS types for point, linestring and polygon should be written as multi-geometries or single geometries. Yes means the table created has multi-geometries, i.e. multipoint, and the features are coerced into multi-geometries. No means the geometry column of the created table is singular, i.e. point, and multi-geometries are split. First-feature allows this setting to be based on what the first feature in the table is. This setting is used for PostGIS-to-PostGIS translations.
<code>fieldName</code>	The name of the field to be written. Valid values for field name include any character string devoid of SQL-offensive characters and less than 32 characters in length.

Parameter	Contents
fieldType	The type of a column in a table. The valid values for the field type are listed below: bool char(width) bpchar(width) varchar(width) int2 int4 int8 text(width) oid serial float4 float8 money date time timetz timestamp timestamptz
indexType	The type of index to create on the given field. The valid values for the index type are listed below: BTREE (default attribute index) RTREE HASH PRIKEY (primary key)

[Workbench Parameter: <WorkbenchParameter>](#)

START_TRANSACTION

Required/Optional: *Optional*

This statement tells the PostGIS writer module when to start actually writing features into the database. The PostGIS writer does not write any features until the feature number of features are skipped, and then it begins writing the following features. Normally, the value specified is zero – a non-zero value is only specified when a data load operation is being resumed after failing partway through.

```
POSTGIS_OUT_START_TRANSACTION 0
```

[Workbench Parameter: <WorkbenchParameter>](#)

TRANSACTION_INTERVAL

Required/Optional: *Optional*

This statement informs the FME about the number of features to be placed in each transaction before a transaction is committed to the database.

If the `POSTGIS_OUT_TRANSACTION_INTERVAL` statement is not specified, then a value of 1000 is used as the transaction interval.

```
POSTGIS_OUT_TRANSACTION_INTERVAL 2000
```

[Workbench Parameter: <WorkbenchParameter>](#)

BULK_COPY

Required/Optional: *Optional*

This statement tells the PostGIS writer module to insert data into the database using either SQL `INSERT` statements or the SQL `COPY` command. The default option is the bulk copy using the `COPY` command, which yields the best performance. The bulk delimiter is no longer user-adjustable – the escaping of it has improved and is no longer necessary as a backup measure. However, if individual inserts are desired, this option can be set to `NO`.

```
POSTGIS_OUT_BULK_COPY YES
```

[Workbench Parameter: <WorkbenchParameter>](#)

WRITER_MODE

Required/Optional: *Optional*

Note: For more information on this directive, see the chapter *Database Writer Mode*.

This directive informs the PostGIS writer which SQL operations will be performed by default by this writer. This operation can be set to `INSERT`, `UPDATE` or `DELETE`. The default writer-level value for this operation can be overwritten at the feature type or table level. The corresponding feature type `DEF` parameter name is called `POSTGIS_MODE`. It has the same valid options as the writer-level mode, as well as the value `INHERIT_FROM_WRITER` (which causes the writer-level mode to be inherited by the feature type as the default for features contained in that table).

The operation can also be set specifically for individual features. Note that when the writer mode is set to `INSERT`, this prevents the mode from being interpreted from individual features and all features are inserted unless otherwise marked as update or delete features. These are skipped.

If the `POSTGIS_OUT_WRITER_MODE` statement is not specified, then a value of `INSERT` is given.

```
POSTGIS_OUT_WRITER_MODE INSERT
```

[Workbench Parameter: <WorkbenchParameter>](#)

GENERIC_GEOMETRY

Required/Optional: *Optional*

This directive is unique in that it only applies at generation time and not at translation time. The default value of `NO` indicates that we want the previous behavior of creating geometrically constrained geometry columns on the destination tables. For example, a `POINT` geometry table would be restricted only to points. Now we have the option to create generic or non-constrained geometry column types.

Effectively this means you can insert multiple geometry types into one table. Specifically the geometry column is created to have the generic type `GEOMETRY` and there are no constraints placed on the geometry types allowed.

If the `POSTGIS_OUT_GENERIC_GEOMETRY` statement is not specified, then a value of `NO` is given.

```
POSTGIS_OUT_GENERIC_GEOMETRY YES
```

[Workbench Parameter: <WorkbenchParameter>](#)

Feature Representation

Features read from PostGIS consist of a series of attribute values and geometry. The feature type of each feature is as defined on its `DEF` line. The geometry object model in PostGIS follows the *OGIS Simple Features Specification 1.1*. For more information see <http://www.opengis.org>. Underlying PostgreSQL geometries are not read as geometries but are interpreted as strings.

Features written to the database have the destination table as their feature type, and attributes as defined on the `DEF` line.

In addition to the generic FME feature attributes that FME Workbench adds to all features (see *About Feature Attributes* on page 7), this format adds the format-specific attributes described in this section.

Attribute Name	Contents
<code>postgis_type</code>	The type of geometric entity stored within the feature. The valid values for the object model are listed below: <code>postgis_point</code> <code>postgis_line</code> <code>postgis_area</code> <code>postgis_geometrycollection</code> <code>postgis_none</code>

Features read from, or written to, PostGIS also have an attribute for each column in the database table. The feature attribute name will be the same as the source or destination column name. The attribute and column names are case-sensitive.

No Coordinates

postgis_type: `postgis_none`

Features with no coordinates are tagged with this value when reading or writing to or from PostGIS. Note that when reading or writing attribute-only data tables, the PostgreSQL reader/writer should be used instead. Note also that this is not a valid OGC type.

Points

postgis_type: `postgis_point`

Features tagged with this value consist of a single point. Both singular points and aggregates of points are supported. Point aggregates will become multipoints while singular points will become points in PostGIS.

Line

postgis_type: postgis_lines

Linear features are tagged with this value when reading or writing to or from PostGIS. A linestring consists of one or more ordered two-point line segments. Line aggregates will become multilinestrings while singular lines will become linestrings in PostGIS.

Area

postgis_type: postgis_area

Area or polygon features are tagged with this value when reading or writing to or from PostGIS. Both single-part and aggregate area features are supported. An area feature may be either a polygon or a donut polygon. Note that checking is done to ensure that the area features adhere to the geometry rules of PostGIS as they are loaded. Area aggregates will become multipolygons while singular areas will become polygons in PostGIS.

GeometryCollections

postgis_type: postgis_geometrycollection

Aggregates containing heterogeneous collections of point, line and polygon features are processed through FME as single features when reading or writing to or from PostGIS. Thus by default no `geometry_collections` are produced.

When reading geometry collection features from PostGIS, the features in the collection are split out into individual FME features and are tagged with the collection number and item number to identify their origin. This results in a less accurate representation of the geometry actually contained in the PostGIS table, but greatly simplifies the processing of such data and its writing to several other formats. When writing back to PostGIS, the `postgis_type` can be manually set to `postgis_geometrycollection` and they will be rewritten as collections.

Geometry

postgis_type: postgis_geometry

Although not a valid geometry type on an individual feature, this type may be set for the destination geometry column type to indicate that any geometry is allowable in that column. If the writer directive `GENERIC_GEOMETRY` is specified at generation time, all destination feature types will have geometry columns of this type. Alternatively, although it will not happen by default, this type can be specified on any one or more destination feature types manually to create generic geometry columns on those specific tables.

Troubleshooting

Problems sometimes arise when attempting to connect to a PostGIS/PostgreSQL database. This is almost always due to a misconfiguration in the user's environment. The following suggestions can often help detect and overcome such problems.

- Ensure you can connect to the database with the host, port, database, user name, and password using `psql` or `pgAdmin`. See PostgreSQL documentation for proper security and connection information, and for the usage of the `psql` utility.
- If you try to list the tables and nothing happens, check the log file. There may have been an underlying error that didn't generate a dialog. Usually this means a parameter does not exist or permissions are not sufficient to access the requested resource.
- In most cases, the `POSTGIS_DATABASE` directive should be left with blank values, with the `POSTGIS_DATASET` directive containing the name of the PostGIS database.
- When using a UNIX operating system, the environment variables `PGHOST`, `PGPORT`, `PGDATABASE`, `PGUSER` and `PGPASSWORD` can be used to specify the PostgreSQL connection parameters.
- If the table list in the PostGIS reader input settings box does not display your table, try the following:
Type the name with the schema prefix, i.e. `public.mytable`. If this works, then your table may not be properly registered in the PostGIS metadata tables or it may not have a geometry column.
- If the table list in the PostGIS reader input settings box lists your table, but you receive an error message that the table does not exist when you run the translation, then it is likely that the PostgreSQL table has been deleted without updating the PostGIS metadata tables. Orphaned metadata may continue to exist in the PostGIS metadata tables. It is suggested that the PostGIS metadata table for the geometry columns be corrected to match only existing PostgreSQL tables.
- If your data ends up looking garbled using a given encoding it may be because the encoding of the data does not match your system encoding. These must match because FME uses the system encoding to set the encoding of the PostgreSQL client, and then allows the database to convert encodings if necessary between the client and server.