

IBM DB2 Reader/Writer

FME's DB2 Database (Attributes only) reader and writer modules (called *DB2 Reader/Writer* throughout the rest of this chapter) provide the Feature Manipulation Engine (FME) with access to attribute data held in IBM's DB2 database tables.

Overview

This data may or may not have a spatial component to it. Thus DB2 reader can read from DB2 databases which may or may not be spatially enabled. The FME provides read and write access to live databases accessible via DB2 CLI.

Tip: See the @SQL function in the *Functions and Factories* manual. This function allows arbitrary Structured Query Language (SQL) statements to be executed against any database.

DB2 Database Quick Facts

Format Type Identifier	DB2
Reader/Writer	Both
Licensing Level	Professional
Dependencies	None
Dataset Type	Data source name
Feature Type	Table name
Typical File Extensions	N/A
Automated Translation Support	Yes
User-Defined Attributes	Yes
Coordinate System Support	No
Generic Color Support	No
Spatial Index	Never
Schema Required	Yes
Transaction Support	Yes
Geometry Type	db2_type

Geometry Support			
Geometry	Supported?	Geometry	Supported?
aggregate	no	point	no
circles	no	polygon	no
circular arc	no	raster	no
donut polygon	no	solid	no
elliptical arc	no	surface	no
ellipses	no	text	no

Geometry Support			
Geometry	Supported?	Geometry	Supported?
line	no	z values	n/a
none	yes		

Reader Overview

FME considers a DB2 dataset to be a collection of relational tables. The tables must be defined in the mapping file before they can be read. Arbitrary `WHERE` clauses and joins are fully supported.

Reader Directives

The suffixes listed are prefixed by the current `<ReaderKeyword>` in a mapping file. By default, the `<ReaderKeyword>` for the DB2 reader is `DB2`.

DATASET

Required/Optional: *Required*

This is the data source name similar to ODBC data source name.

Example:

```
DB2_DATASET sample
```

Workbench Parameter: [<WorkbenchParameter>](#)

USER_NAME

Required/Optional: *Optional*

The name of the user who will access the database. By default, `USER_NAME` will be considered the same as the schema name. e.g. any table name which explicitly does not have the schema name prefixed will be considered as a table from the schema for that user.

Example:

```
DB2_USER_NAME bond007
```

Workbench Parameter: [<WorkbenchParameter>](#)

PASSWORD

Required/Optional: *Optional*

The password to access the database.

Example:

```
DB2_PASSWORD moneypenny
```

Workbench Parameter: [<WorkbenchParameter>](#)

DEF**Required/Optional:** *Optional*

Each database table must be defined before it can be read. There are two forms that the definition may take.

The syntax of the first form is:

```
DB2_DEF <tableName>                                     \
    [SQL_WHERE_CLAUSE <whereClause>]                   \
    [<fieldName> <fieldType>] +
```

In this form, the fields and their types are listed. 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 <tableName> must match a table in the database. This will be used as the feature type of all the features read from the table.

If no <whereClause> is specified, all rows in the table will be read and returned as individual features, unless limited by a global directive:

```
<ReaderKeyword>_WHERE_CLAUSE
```

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."

In this example, the all records whose ID is less than 5 will be read from the supplier table:

```
DB2_DEF supplier                                       \
    SQL_WHERE_CLAUSE "id < 5"                         \
    ID integer                                         \
    NAME char(100)                                     \
    CITY char(50)
```

The syntax of the second form is:

```
DB2_DEF <tableName>                                     \
    SQL_STATEMENT <sqlStatement>
```

In this form, an arbitrary complete <sqlStatement> will be executed. The statement is passed untouched to the database (and therefore may include non-portable database constructions). The results of the statement will be returned, one row at a time, as features to FME. This form allows the results of complex joins to be returned to FME.

Note: If the table has a column of type BIGINT then use the DB2's CHAR() function to convert it to a string. This also applies when an arbitrary SQL statement is passed to FME using @SQL() function or the SQLExecutor transformer in Workbench. For example,

```
SELECT CHAR(myBigIntColumn), myID FROM myTable
```

All features will be given the feature type <tableName>, even though they may not necessarily have come from that particular table. Indeed, with this form, the <tableName> need not exist as a separate table in the database.

In this example, the results of joining the `employee` and `city` tables are returned. All attributes from the two tables will be present on each returned feature. The feature type will be set to `complex`.

```
DB2_DEF complex \
  SQL_STATEMENT \
  "SELECT * FROM EMPLOYEE, CITY WHERE EMPLOYEE.CITY = CITY.NAME"
```

Workbench Parameter: [<WorkbenchParameter>](#)

WHERE_CLAUSE

Required/Optional: *Optional*

This optional specification is used to limit the rows read by the reader from each table. If a given table has no `SQL_WHERE_CLAUSE` or `SQL_STATEMENT` specified in its `DEF` line, the global `<ReaderKeyword>_WHERE_CLAUSE` value, if present, will be applied as the `WHERE` specifier of the query used to generate the results. If a table's `DEF` line does contain its own `SQL_WHERE_CLAUSE` or `SQL_STATEMENT`, it will override the global `WHERE` clause.

The syntax for this clause is:

```
DB2_WHERE_CLAUSE <whereClause>
```

Note that the `<whereClause>` does not include the word "WHERE."

The example below selects only the features whose lengths are more than 2000:

```
DB2_WHERE_CLAUSE LENGTH > 2000
```

Workbench Parameter: [<WorkbenchParameter>](#)

IDs

Required/Optional: *Optional*

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

```
DB2_IDS <featureType1> \
<featureType2> ... \
<featureTypeN>
```

The feature types must match those used in `DEF` lines.

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

```
DB2_IDS HISTORY
```

Workbench Parameter: [<WorkbenchParameter>](#)

RETRIEVE_ALL_SCHEMAS

Required/Optional: *Optional*

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

This optional directive 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, it is assumed to be "No".

The syntax of the `RETRIEVE_ALL_SCHEMAS` directive is:

```
DB2_RETRIEVE_ALL_SCHEMAS Yes
```

RETRIEVE_ALL_TABLE_NAMES

Required/Optional: *Optional*

This directive 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 directive is used to tell the reader to only retrieve the table names of all the tables in the source database. If `RETRIEVE_ALL_SCHEMAS` is also set to "Yes", then `RETRIEVE_ALL_SCHEMAS` will take precedence. If this value is not specified, it is assumed to be "No".

The syntax of the `RETRIEVE_ALL_TABLE_NAMES` directive is:

```
DB2_RETRIEVE_ALL_TABLE_NAMES Yes
```

Writer Overview

The Database writer module stores attribute records into a live relational database. The Database writer provides the following capabilities:

- **Transaction Support:** The Database writer provides transaction support that eases the data loading process. Occasionally, a data load operation terminates prematurely due to data difficulties. The transaction support provides a mechanism for reloading corrected data without data loss or duplication.
- **Table Creation:** The Database writer uses the information within the FME mapping file to automatically create database tables as needed.
- **Bulk Loading:** The Database writer uses a bulk loading technique to ensure speedy data load only when there are no LOB (BLOBs or CLOBs) columns in the table. The performance vastly exceeds a one-insert-at-a-time approach.

Writer Directives

The directives processed by the DB2 Writer are listed below. The suffixes shown are prefixed by the current `<WriterKeyword>` in a mapping file. By default, the `<WriterKeyword>` for the DB2 writer is `DB2`.

DATASET, USER_NAME, PASSWORD

The `DATASET`, `USER_NAME`, and `PASSWORD` directives operate in the same manner as they do for the DB2 reader. The remaining writer-specific directives are discussed in the following sections.

ABORT_ON_BAD_DATA

Required/Optional: *Optional*

Some features may contain out-of-range or invalid attribute values. These features will be rejected and cannot be written to the database. If the value of this directive is YES then the translation will be aborted immediately after encountering such a problem. If this directive is set to NO then the translation will continue but the features with rejected feature will not be written to the database.

Values: YES | NO

Default: NO

Example:

```
DB2_ABORT_ON_BAD_DATA YES
```

DEF

Required/Optional: *Optional*

Each database table must be defined before it can be written. For the DB2 writer, only one form of the DEF line is used:

```
DB2_DEF <tableName>                                     \
      [db2_overwrite_table      (YES|NO|TRUNCATE) ]      \
      [<fieldName>             <fieldType>] +
```

In this form, the fields and their types are listed. If the table already exists in the database, and `db2_overwrite_table` is not specified with a parameter of YES, FME will append its information the existing database table. In this case, it is not necessary to list the fields and their types – FME will use the schema information in the database to determine this. If the fields and types are listed, they must match those in the database. However, not all fields must be listed.

If the table does not exist, or `db2_overwrite_table` is specified with a value of YES, 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.

This example defines the SUPPLIER table for the FME. If the table did not exist, it will be created just before the first SUPPLIER row is written. If the table already exists, the data will be appended to the existing table.

```
DB2_DEF SUPPLIER                                       \
      ID integer                                       \
      NAME char(100)                                   \
      CITY char(50)
```

The following example is exactly the same, except that it replaces any existing table named SUPPLIER with a new table having the specified definition. If the table SUPPLIER does not exist in the database, then a new table is simply created.

```
DB2_DEF SUPPLIER                                       \
      db2_overwrite_table YES                          \
      ID integer                                       \
      NAME char(100)                                   \
      CITY char(50)
```

In the following example, the definition line only make the pre-existing `EMPLOYEE` table known to FME:

```
DB2_DEF EMPLOYEE
```

Features may later be routed to this table.

TRANSACTION_INTERVAL

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

If the `DB2_TRANSACTION_INTERVAL` statement is not specified, then a value of 1000 is used as the transaction interval.

Parameter	Contents
<transaction_interval>	The number of features in a single transaction.

Example:

```
DB2_TRANSACTION_INTERVAL 5000
```

Feature Representation

Features read from a DB2 database consist of a series of attribute values. They have no geometry. The attribute names are as defined in the `DEF` line if the first form of the `DEF` line was used. If the second form of the `DEF` line was used, then the attribute names are as they are returned by the query, and as such may have their original table names as qualifiers. The feature type of each DB2 feature is as defined on its `DEF` line.

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

DATE, TIME and DATETIME Fields

When a `DATE`, `TIME` or `TIMESTAMP` field is read by the DB2 reader, two attributes are set in the FME feature. The first attribute is has the name of the database column, and its value is of the form `YYYYMMDD` or `HHMMSS`. This is compatible with all other FME date and time values.

The second attribute has a suffix of `.full` and is of the form `YYYYMMDDHHMMSS`. It specifies the date and the time, with the time portion specified using the 24-hour clock.

For example, if a date field called `UPDATE_DATE` is read, the following attributes will be set in the retrieved FME feature:

```
UPDATE_DATE = '19980820'
UPDATE_DATE.full = '19980820000000'
```

The DB2 writer looks for both attributes when a `date` or `datetime` column is being output. Either may be specified. If both attributes are specified, then the value specified in `UPDATE_DATE.full` is used to populate the `DATE` or `DATETIME` portion of the date; otherwise, this portion is set to 0.

Using DEF Lines to Read from an ODBC Datasource

This example illustrates how the two forms of the DEF lines can be used to read from an ODBC database source, which is named `rogers`.

```

READER_TYPE DB2
DB2_DATASET      sampledb
DB2_USER_NAME    <userName>
DB2_PASSWORD     <password>

# Form 1 of the DEF line is used like this -- it reads just
# the two fields we list and applies the where clause

DB2_DEF supplier                                     \
    db2_where_clause "id < 5"                       \
    ID integer                                       \
    CITY char(50)

# Form 2 of the DEF line is used like this -- we let SQL
# figure out what fields we want and do a complex join
# involving 3 tables. The FME features will have whatever
# fields are relevant. The "feature type" as far as
# FME is concerned is whatever was put on the DEF line.
# In this case "complex" is the feature type, even though no
# table named "complex" is present in the database.

DB2_DEF complex                                     \
    db2_sql "SELECT CUSTOMER.NAME, CUSTOMER.ID,
    VIDEOS.ID, VIDEOS.TITLE FROM RENTALS, CUSTOMER,
    VIDEOS WHERE RENTALS.customerID = CUSTOMER.ID AND
    VIDEOS.ID = RENTALS.videoID AND CUSTOMER.ID = 1"

# Finally, define the NULL writer as our output -- we will
# just log everything we read to the log file for inspection.

WRITER_TYPE NULL
NULL_DATASET null

FACTORY_DEF * SamplingFactory                       \
    INPUT FEATURE_TYPE * @Log()

```