

SQLite Reader/Writer

Overview

The SQLite reader and writer modules provide FME with access to attribute data held in sqlite3 database tables. This data may not necessarily have a spatial component to it. FME provides read and write access to sqlite3 databases.

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

SQLite Quick Facts

Format Type Identifier	SQLITE3
Reader/Writer	Both
Licensing Level	Base
Dependencies	None
Dataset Type	Database
Feature Type	Table name
Typical File Extensions	.db .sl3
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
Encoding Support	Yes
Geometry Type	db_none

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
line	no	z values	n/a
none	yes		

Reader Overview

FME considers a database data set 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 SQLite3 reader is `SQLITE3`.

DATASET

Required/Optional: *Required*

This is the file name of the SQLite3 Database.

For example,

```
SQLITE3_DATASET c:/data/citySource.db
```

DEF

Required/Optional: *Required*

The syntax of the definition is:

```
SQLITE3_DEF <tableName>                                     \
[sqlite3_sql_statement <sqlQuery>]                         \
  [sqlite3_where_clause<whereClause>]                       \
[<fieldName>      <fieldType>] +
```

OR

```
SQLITE3_DEF <queryName>                                     \
  [sqlite3_sql_statement <sqlQuery>]                         \
```

The <tableName> must match the name of an existing SQLite3 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 `sqlite3_sql_statement` keyword. 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 definition allows specification of separate search parameters for each table. If any of the per table configuration parameters are given, they will override, for that table, whatever global values have been specified by the reader keywords such as the `WHERE_CLAUSE`. 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>sqlite3_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 rows are returned. This keyword will be ignored if the <code>sql3_sql_statement</code> is present.
<code>sqlite3_sql_statement</code>	This specifies an SQL SELECT query to be used as the source for the results. If this is specified, the SQLite3 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. The <code>sqlite3_where_clause</code> is ignored if <code>sqlite3_sql_statement</code> is supplied. This form allows the results of complex joins to be returned to FME.

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 SQLite3 reader allows one to use the `sqlite3_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 at least one 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>`.

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

```
SQLITE3_DEF supplier \
  sqlite3_where_clause "id < 5" \
  ID integer \
  NAME text \
  CITY text
```

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`.

```
SQLITE3_DEF complex \
  sqlite3_sql_statement \
  "SELECT * FROM EMPLOYEE, CITY WHERE EMPLOYEE.CITY = CITY.NAME"
```

IDs

Required/Optional: *Optional*

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

```

SQLITE3_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:

```
SQLITE3_IDS HISTORY
```

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.

When set to "Yes", indicates to the reader to return all the schemas of the tables in the database.

If this value is not specified, it is assumed to be "No".

Range: YES | NO

Default: NO

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 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".

Range: YES | NO

Default: NO

Writer Overview

The SQLite3 writer module stores attribute records into a live relational database. The SQLite3 writer provides the following capabilities:

- **Transaction Support:** The SQLite3 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 SQLite3 writer uses the information within the FME mapping file to automatically create database tables as needed.
- **Writer Mode Specification:** The SQLite3 writer allows the user to specify what database command should be issued for each feature received. Valid writer modes are INSERT, UPDATE and DELETE. The writer mode can be specified at three unique levels: at the writer level, on the feature type, or on individual features.

Writer Directives

The directives listed below are processed by the SQLite3 writer. The suffixes shown are prefixed by the current `<WriterKeyword>` in a mapping file. By default, the `<WriterKeyword>` for the SQLite3 writer is `SQLITE3`.

DATASET

Required/Optional: *Required*

The `DATASET` directive operates in the same manner as it does for the SQLite3 reader.

DEF

Required/Optional: *Required*

Each SQLite3 table must be defined before it can be written. The general form of a SQLite3 definition statement is:

```
SQLITE3_DEF <tableName>                                     \
  [sqlite3_update_key_columns <keyColumns>]                \
  [sqlite3_drop_table      (yes|no)]                       \
  [sqlite3_truncate_table  (yes|no)]                       \
  [sqlite3_table_writer_mode (inherit_from_writer|insert| \
  update|delete)]                                         \
  [<fieldName>      <fieldType>[, <indexType>]]+
```

The table definition allows control of the table that will be created. If the fields and types are listed, the types must match those in the database. Fields which can contain NULL values do not need to be listed - these fields will be filled with NULL values.

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
tableName	The name of the table to be written. If a table with the specified name exists, it will be overwritten if the <code>sqlite3_drop_table</code> DEF line parameter is set to <code>YES</code> , or it will be truncated if the <code>sqlite3_truncate_table</code> DEF line parameter 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 (the <code>"</code> is the only SQL-offensive character in SQLite) and less than 255 characters in length.

Parameter	Contents
<code>sqlite3_table_writer_mode</code>	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>sqlite3_update_key_columns</code>	This is a comma-separated list of the columns which are matched against the corresponding FME attributes' values to specify which rows are to be updated or deleted when the writer mode is either <code>UPDATE</code> or <code>INSERT</code> . For example: <code>sqlite3_update_key_columns ID</code> would instruct the writer to ensure that the ID attribute is always matched against the column with the same name. Also, the target table is always the feature type specified in the DEF line. Each column listed with the <code>sqlite3_update_key_columns</code> keyword must be defined with a type on the DEF line, in addition to the columns whose values will be updated by the operation.
<code>sqlite3_drop_table</code>	This specifies that if the table exists by this name, it should be dropped and replaced with a table specified by this definition. Default: <code>NO</code>
<code>sqlite3_truncate_table</code>	This specifies that if the table exists by this name, it should be cleared prior to writing. Default: <code>NO</code>
<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 (the <code>"</code> is the only SQL-offensive character in SQLite) and less than 255 characters in length.
<code>fieldType</code>	The type of a column in a table. The valid values for the field type are listed below: <code>blob</code> <code>float</code> <code>integer</code> <code>real(width, decimal)</code> <code>text</code> <code>varchar(width)</code>
<code>indexType</code>	The type of index to create for the column. If the table does not previously exist, then upon table creation, a database index of the specified type is created. The database index contains only the one column. The valid values for the column type are listed below: <code>indexed</code> : An index without constraints. <code>unique</code> : An index with a unique constraint.

START_TRANSACTION**Required/Optional:** *Optional*

This statement tells the SQLite3 writer module when to start actually writing features into the database. The SQLite3 writer does not write any features until the feature is reached that belongs to `<last successful transaction> + 1`. Specifying a value of zero causes every feature to be output. 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.

Parameter	Contents
<code><last successful transaction></code>	The transaction number of the last successful transaction. When loading data for the first time, set this value to 0. Default: 0

Example:

```
SQLITE3_START_TRANSACTION 0
```

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 `SQLITE3_TRANSACTION_INTERVAL` statement is not specified, then a value of 500 is used as the transaction interval.

Parameter	Contents
<code><transaction_interval></code>	The number of features in a single transaction. Default: 500

If the `SQLITE3_TRANSACTION_INTERVAL` is set to zero, then feature based transactions are used. As each feature is processed by the writer, they are checked for an attribute called `fme_db_transaction`. The value of this attribute specifies whether the writer should commit or rollback the current transaction. The value of the attribute can be one of `COMMIT_BEFORE`, `COMMIT_AFTER`, `ROLLBACK_AFTER` or `IGNORE`. If the `fme_db_transaction` attribute is not set in any features, then the entire write operation occurs in a single transaction.

Example:

```
SQLITE3_TRANSACTION_INTERVAL 5000
```

WRITER_MODE

Required/Optional: *Optional*

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

This directive informs the SQLite3 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

`sqlite3_table_writer_mode`. It has the same valid options as the writer level mode and additionally 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 be set specifically for individual features as well. 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 `SQLITE3_WRITER_MODE` statement is not specified, then a value of `INSERT` is given.

Parameter	Contents
<code><writer_mode></code>	The type of SQL operation that should be performed by the writer. The valid list of values are below: INSERT UPDATE DELETE Default: INSERT

Example:

```
SQLITE3_WRITER_MODE INSERT
```

BEGIN_SQL{n}

Required/Optional: *Optional*

Occasionally, you must execute some arbitrary SQL statements before opening an SQLite3 table.

Upon opening a connection to read from an SQLite3 database, the SQLite3 writer looks for the directive `<ReaderKeyword>_BEGIN_SQL{n}` (for $n=0, 1, 2, \dots$), and executes each such directive's value as an SQL statement on the database connection.

Multiple SQL commands can be delimited by a character specified using the `FME_SQL_DELIMITER` keyword, embedded at the beginning of the SQL block. The single character following this keyword will be used to split the SQL, which will then be sent to the database for execution. **Note:** Include a space before the character.

For example:

```
FME_SQL_DELIMITER ;
DELETE FROM instructors;
DELETE FROM people;
WHERE LastName='Doe' AND FirstName='John'
```

Any errors occurring during the execution of these SQL statements will normally terminate the writer with an error. If the specified statement is preceded by a hyphen ("-"), such errors are ignored.

END_SQL{n}

Required/Optional: *Optional*

Occasionally one must execute some arbitrary SQL statements after closing a set of SQLite3 tables.

Just prior to closing a connection on an SQLite3 database, the SQLite3 writer looks for the directive `<WriterKeyword>_END_SQL{n}` (for $n=0,1,2,\dots$), and executes each such directive's value as an SQL statement on the database connection.

Multiple SQL commands are allowed on a single `END_SQL` line. This can be done by separating each command with a semicolon. If the SQL statement has literals or identifiers that also contain semicolons the statement may be parsed incorrectly. To avoid this problem please either use multiple `END_SQL` lines or avoid the use of embedded semicolons if possible.

Any errors occurring during the execution of these SQL statements will normally terminate the writer with an error. If the specified statement is preceded by a hyphen ("-"), such errors are ignored.

INIT_TABLES

Required/Optional: *Optional*

This directive informs the SQLite3 writer when each table should be initialized. Initialization encompasses the actions of dropping or truncating existing tables, and creating new tables as necessary.

When `INIT_TABLES` is set to `IMMEDIATELY`, the SQLite3 writer will initialize all tables immediately after parsing the `DEF` lines and opening the database file. In this mode, all tables will be initialized, even if the SQLite3 writer receives no features for a given table.

When `INIT_TABLES` is set to `FIRSTFEATURE`, the SQLite3 writer will only initialize a table once the first feature destined for that table is received. In this mode, if the SQLite3 writer does not receive any features for a given table, the table will never be initialized.

Writer Mode Specification

The SQLite3 writer allows the user to specify a writer mode, which determines what database command should be issued for each feature received. Valid writer modes are `INSERT`, `UPDATE` and `DELETE`.

Writer Modes

In `INSERT` mode, the attribute values of each received feature are written as a new database record.

In `UPDATE` mode, the attribute values of each received feature are used to update existing records in the database. The records which are updated are determined via the `sqlite3_update_key_columns DEF` line parameter, or via the `fme_where` attribute on the feature.

In `DELETE` mode, existing database records are deleted according to the information specified in the received feature. Records are selected for deletion using the same technique as records are selected for updating in `UPDATE` mode.

Writer Mode Constraints

In UPDATE and DELETE mode, the `fme_where` attribute always takes precedence over the `sqlite3_update_key_columns` DEF line parameter. If both the `fme_where` attribute and the `sqlite3_update_key_columns` DEF line parameter are not present, then UPDATE or DELETE mode will generate an error.

When the `fme_where` attribute is present, it is used verbatim as the WHERE clause on the generated UPDATE or DELETE command. For example, if `fme_where` were set to `'id<5'`, then all database records with field ID less than 5 will be affected by the command.

When the `fme_where` attribute is not present, the writer looks for the `sqlite3_update_key_columns` DEF line parameter and uses it to determine which records should be affected by the command. Please refer to DEF for more information about the `sqlite3_update_key_columns` DEF line parameter.

Writer Mode Selection

The writer mode can be specified at three unique levels. It may be specified on the writer level, on the feature type or on individual features.

At the writer level, the writer mode is specified by the `WRITER_MODE` keyword. This keyword can be superseded by the feature type writer mode specification. **Note:** For more information on this directive, see the chapter *Database Writer Mode*.

At the feature type level, the writer mode is specified by the `sqlite3_writer_mode` DEF line parameter. This parameter supersedes the `WRITER_MODE` keyword. Unless this parameter is set to `INSERT`, it may be superseded on individual features by the `fme_db_operation` attribute. Please refer to the DEF line documentation for more information about this parameter.

At the feature level, the writer mode is specified by the `fme_db_operation` attribute. Unless the parameter at the feature type level is set to `INSERT`, the writer mode specified by this attribute always supersedes all other values. Accepted values for the `fme_db_operation` attribute are `INSERT`, `UPDATE` or `DELETE`.

Feature Representation

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

Features read from a 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 SQLite3 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.