SQLite SQL dialect

Since GDAL/OGR 1.10, the SQLite "dialect" can be used as an alternate SQL dialect to the OGR SQL dialect. This assumes that GDAL/OGR is built with support for SQLite (>= 3.6), and preferably with Spatialite support too to benefit from spatial functions.

The SQLite dialect may be used with any OGR datasource, like the OGR SQL dialect. It is available through the **GDALDataset::ExecuteSQL()** method by specifying the pszDialect to "SQLITE". For the ogrinfo or ogr2ogr utility, you must specify the "-dialect SQLITE" option.

This is mainly aimed to execute SELECT statements, but, for datasources that support update, INSERT/UPDATE/DELETE statements can also be run.

The syntax of the SQL statements is fully the one of the SQLite SQL engine. You can refer to the following pages:

- SELECT documentation
- INSERT documentation
- UPDATE documentation
- DELETE documentation

SELECT statement

The SELECT statement is used to fetch layer features (analogous to table rows in an RDBMS) with the result of the query represented as a temporary layer of features. The layers of the datasource are analogous to tables in an RDBMS and feature attributes are analogous to column values. The simplest form of OGR SQLITE SELECT statement looks like this:

```
SELECT * FROM polylayer
```

More complex statements can of course be used, including WHERE, JOIN, USING, GROUP BY, ORDER BY, sub SELECT, ...

The table names that can be used are the layer names available in the datasource on which the ExecuteSQL() method is called.

Similarly to OGRSQL, it is also possible to refer to layers of other datasources with the following syntax : "other_datasource_name"."layer_name".

```
SELECT p.*, NAME FROM poly p JOIN "idlink.dbf"."idlink" il USING (eas_id)
```

The column names that can be used in the result column list, in WHERE, JOIN, ... clauses are the field names of the layers. Expressions, SQLite functions can also be used, spatial functions, etc...

The conditions on fields expressed in WHERE clauses, or in JOINs are translated, as far as possible, as attribute filters that are applied on the underlying OGR layers. Joins can be very expensive operations if the secondary table is not indexed on the key field being used.

Delimited identifiers

If names of layers or attributes are reserved keywords in SQL like 'FROM' or they begin with a number or underscore they must be handled as "delimited identifiers" and enclosed between double quotation marks in queries. Double quotas can be used even when they are not strictly needed.

```
SELECT "p"."geometry", "p"."FROM", "p"."3D" FROM "poly" p
```

When SQL statements are used in the command shell and the statement itself is put between double quotes, the internal double quotes must be escaped with \

```
ogrinfo p.shp -sql "SELECT geometry \"FROM\", \"3D\" FROM p"
```

Geometry field

The **GEOMETRY** special field represents the geometry of the feature returned by **OGRFeature::GetGeometryRef()**. It can be explicitly specified in the result column list of a SELECT, and is automatically selected if the wildcard is used.

For OGR layers that have a non-empty geometry column name (generally for RDBMS datasources), as returned by **OGRLayer::GetGeometryColumn()**, the name of the geometry special field in the SQL statement will be the name of the geometry column of the underlying OGR layer.

```
SELECT EAS_ID, GEOMETRY FROM poly

returns:

OGRFeature(SELECT):0
  EAS_ID (Real) = 168
  POLYGON ((479819.84375 4765180.5,479690.1875 4765259.5,[...],479819.84375 4765180.5))
```

```
SELECT * FROM poly

returns:

OGRFeature(SELECT):0
  AREA (Real) = 215229.266
  EAS_ID (Real) = 168
  PRFEDEA (String) = 35043411
  POLYGON ((479819.84375 4765180.5,479690.1875 4765259.5,[...],479819.84375 4765180.5))
```

OGR_STYLE special field

The **OGR_STYLE** special field represents the style string of the feature returned by **OGRFeature::GetStyleString()**. By using this field and the **LIKE** operator the result of the query can be filtered by the style. For example we can select the annotation features as:

```
SELECT * FROM nation WHERE OGR_STYLE LIKE 'LABEL%'
```

Spatialite SQL functions

When GDAL/OGR is build with support for the Spatialite library, a lot of extra SQL functions, in particular spatial functions, can be used in results column fields, WHERE clauses, etc....

```
SELECT EAS_ID, ST_Area(GEOMETRY) AS area FROM poly WHERE
    ST_Intersects(GEOMETRY, BuildCircleMbr(479750.6875,4764702.0,100))

returns:

OGRFeature(SELECT):0
  EAS_ID (Real) = 169
  area (Real) = 101429.9765625
```

```
OGRFeature(SELECT):1
  EAS_ID (Real) = 165
  area (Real) = 596610.3359375

OGRFeature(SELECT):2
  EAS_ID (Real) = 170
  area (Real) = 5268.8125
```

OGR datasource SQL functions

The **ogr_datasource_load_layers(datasource_name[, update_mode[, prefix]])** function can be used to automatically load all the layers of a datasource as VirtualOGR tables.

```
sqlite> SELECT load_extension('libgdal.so');

sqlite> SELECT load_extension('libspatialite.so');

sqlite> SELECT ogr_datasource_load_layers('poly.shp');
1
sqlite> SELECT * FROM sqlite_master;
table|poly|poly|0|CREATE VIRTUAL TABLE "poly" USING VirtualOGR('poly.shp', 0, 'poly')
```

OGR layer SQL functions

The following SQL functions are available and operate on a layer name : **ogr_layer_Extent()**, **ogr_layer_SRID()**, **ogr_layer_GeometryType()** and **ogr_layer_FeatureCount()**

```
SELECT ogr_layer_Extent('poly'), ogr_layer_SRID('poly') AS srid,
       ogr_layer_GeometryType('poly') AS geomtype, ogr_layer_FeatureCount('poly') AS count

returns:

OGRFeature(SELECT):0
  srid (Integer) = 40004
  geomtype (String) = POLYGON
  count (Integer) = 10
  POLYGON ((478315.53125 4762880.5,481645.3125 4762880.5,481645.3125 4765610.5,478315.53125
        4765610.5,478315.53125 4762880.5))
```

OGR compression functions

ogr_deflate(text_or_blob[, compression_level]) returns a binary blob compressed with the ZLib deflate algorithm. See **CPLZLibDeflate()**

ogr_inflate(compressed_blob) returns the decompressed binary blob, from a blob compressed with the ZLib deflate algorithm. If the decompressed binary is a string, use CAST(ogr_inflate(compressed_blob) AS VARCHAR). See **CPLZLibInflate()**.

Other functions

Starting with OGR 2.0, the *hstore_get_value()* function can be used to extract a value associate to a key from a HSTORE string, formatted like "key=>value,other_key=>other_value,..."

```
SELECT hstore_get_value('a => b, "key with space"=> "value with space"', 'key with space') --> 'value
        with space'
```

OGR geocoding functions

The following SQL functions are available : **ogr_geocode(...)** and **ogr_geocode_reverse(...)**.

ogr_geocode(name_to_geocode [, field_to_return [, option1 [, option2, ...]]]) where name_to_geocode is a literal or a column name that must be geocoded. field_to_return if specified can be "geometry" for the geometry (default), or a field name of the layer returned by **OGRGeocode()**. The special field "raw" can also be used to return the raw response (XML string) of the geocoding service. option1, option2, etc.. must be of the key=value format, and are options understood by **OGRGeocodeCreateSession()** or **OGRGeocode()**.

This function internally uses the **OGRGeocode()** API. Refer to it for more details.

```
SELECT ST_Centroid(ogr_geocode('Paris'))

returns:

OGRFeature(SELECT):0
  POINT (2.342878767069653 48.85661793020374)
```

```
ogrinfo cities.csv -dialect sqlite -sql "SELECT *, ogr_geocode(city, 'country') AS country,
        ST_Centroid(ogr_geocode(city)) FROM cities"

returns:

OGRFeature(SELECT):0
  id (Real) = 1
  city (String) = Paris
  country (String) = France métropolitaine
  POINT (2.342878767069653 48.85661793020374)

OGRFeature(SELECT):1
  id (Real) = 2
  city (String) = London
  country (String) = United Kingdom
  POINT (-0.109369427546499 51.500506667319407)

OGRFeature(SELECT):2
  id (Real) = 3
  city (String) = Rennes
  country (String) = France métropolitaine
  POINT (-1.68185153381778 48.111663929761093)

OGRFeature(SELECT):3
  id (Real) = 4
  city (String) = Strasbourg
  country (String) = France métropolitaine
  POINT (7.767762859150757 48.571233274141846)

OGRFeature(SELECT):4
  id (Real) = 5
  city (String) = New York
  country (String) = United States of America
  POINT (-73.938140243499049 40.663799577449979)

OGRFeature(SELECT):5
  id (Real) = 6
  city (String) = Berlin
  country (String) = Deutschland
  POINT (13.402306623451983 52.501470321410636)

OGRFeature(SELECT):6
  id (Real) = 7
  city (String) = Beijing
  country (String) = 中华人民共和国
  POINT (116.391195 39.9064702)

OGRFeature(SELECT):7
  id (Real) = 8
  city (String) = Brasilia
  country (String) = Brasil
  POINT (-52.830435216371839 -10.828214867369699)

OGRFeature(SELECT):8
  id (Real) = 9
  city (String) = Moscow
  country (String) = Российская Федерация
  POINT (37.367988106866868 55.556208255649558)
```

ogr_geocode_reverse(longitude, latitude, field_to_return [, option1 [, option2, ...]]) where longitude, latitude is the coordinate to query. field_to_return must be a field name of the layer returned by **OGRGeocodeReverse()** (for example 'display_name'). The special field "raw" can also be used to return the raw response (XML string) of the geocoding service. option1, option2, etc.. must be of the key=value format, and are options understood by **OGRGeocodeCreateSession()** or **OGRGeocodeReverse()**.

ogr_geocode_reverse(geometry, field_to_return [, option1 [, option2, ...]]) is also accepted as an alternate syntax where geometry is a (Spatialite) point geometry.

This function internally uses the **OGRGeocodeReverse()** API. Refer to it for more details.

Spatialite spatial index

Spatialite spatial index mechanism can be triggered by making sure a spatial index virtual table is mentioned in the SQL (of the form idx_layername_geometrycolumn), or by using the more recent SpatialIndex from the VirtualSpatialIndex extension. In which case, a in-memory RTree will be built to be used to speed up the spatial queries.

For example, a spatial intersection between 2 layers, by using a spatial index on one of the layers to limit the number of actual geometry intersection computations :

```
SELECT city_name, region_name FROM cities, regions WHERE
    ST_Area(ST_Intersection(cities.geometry, regions.geometry)) > 0 AND
    regions.rowid IN (
        SELECT pkid FROM idx_regions_geometry WHERE
            xmax >= MbrMinX(cities.geometry) AND xmin <= MbrMaxX(cities.geometry) AND
            ymax >= MbrMinY(cities.geometry) AND ymin <= MbrMaxY(cities.geometry))
```

or more elegantly :

```
SELECT city_name, region_name FROM cities, regions WHERE
    ST_Area(ST_Intersection(cities.geometry, regions.geometry)) > 0 AND
    regions.rowid IN (
        SELECT rowid FROM SpatialIndex WHERE
            f_table_name = 'regions' AND search_frame = cities.geometry)
```