PostgreSQL OGR Foreign Data Wrapper

Motivation

OGR is the **vector** half of the GDAL spatial data access library. It allows access to a large number of GIS data formats using a simple C API for data reading and writing. Since OGR exposes a simple table structure and PostgreSQL foreign data wrappers allow access to table structures, the fit seems pretty perfect.

Limitations

This implementation currently has the following limitations:

- **PostgreSQL 11 or higher.**
- **Limited non-spatial query restrictions are pushed down to OGR.** OGR only supports a minimal set of SQL operators (>, <, <=, >=, =).
- **Only bounding box filters (&&) are pushed down.** Spatial filtering is possible, but only bounding boxes, and only using the && operator.
- **OGR connections every time** Rather than pooling OGR connections, each query makes (and disposes of) **two** new ones, which seems to be the largest performance drag at the moment for restricted (small) queries.
- **All columns are retrieved every time.** PostgreSQL foreign data wrappers don't require all columns all the time, and some efficiencies can be gained by only requesting the columns needed to fulfill a query. This would be a minimal efficiency improvement, but can be removed given some development time, since the OGR API supports returning a subset of columns.

Download

- Windows
  - Via Stackbuilder (part of PostGIS Bundle)
- Linux
  - Arch Linux
  - Ubuntu, PGDG Apt: Debian / Ubuntu
  - PGDG Yum: Redhat EL, CentOS, Rocky
- OSX

Basic Operation

In order to access geometry data from OGR, the PostGIS extension has to be installed: if it is not installed, geometry will be represented as bytea columns, with well-known binary (WKB) values.

To build the wrapper, make sure you have the GDAL library and development packages (is `gdal-config` on your path?) installed, as well as the PostgreSQL development packages (is `pg_config` on your path?)

Build the wrapper with `make` and `make install`. Now you are ready to create a foreign table.

First install the `postgis` and `ogr_fdw` extensions in your database.

```
-- Install the required extensions
CREATE EXTENSION postgis;
CREATE EXTENSION ogr_fdw;
```

For a test data set, copy the `pt_two` example shape file from the `data` directory to a location where the PostgreSQL server can read it (like `/tmp/test/` for example).

Use the `ogr_fdw_info` tool to read an OGR data source and output a server and table definition for a particular layer. (You can write these manually, but the utility makes it a little more foolproof.)

```
# ogr_fdw_info -f

Supported Formats:
    -> "PCIDSK" (read/write)
    -> "netCDF" (read/write)
    ...
    -> "HTTP" (readonly)

# ogr_fdw_info -s /tmp/test

Layers:
    pt_two

# ogr_fdw_info -s /tmp/test -l pt_two

CREATE SERVER "myserver"
    FOREIGN DATA WRAPPER ogr_fdw
    OPTIONS (
        datasource '/tmp/test',
        format 'ESRI Shapefile' );

CREATE FOREIGN TABLE "pt_two" (
    fid integer,
    "geom" geometry(Point, 4326),
    "name" varchar,
    "age" integer,
    "height" real,
    "birthdate" date )
    SERVER "myserver"
    OPTIONS (layer 'pt_two');
```

Copy the `CREATE SERVER` and `CREATE FOREIGN SERVER` SQL commands into the database and you'll have your foreign table definition.

```
                         Foreign table "public.pt_two"
    Column  |       Type        | Modifiers | FDW Options
----------+-------------------+-----------+-------------
 fid      | integer           |           |
 geom     | geometry          |           |
 name     | character varying |           |
 age      | integer           |           |
 height   | real              |           |
 birthday | date              |           |

Server: tmp_shape
FDW Options: (layer 'pt_two')
```

And you can query the table directly, even though it's really just a shape file.

```sql
SELECT * FROM pt_two;
```

```
 fid |                    geom                    | name  | age | height |  birthday
-----+--------------------------------------------+-------+-----+--------+------------
     0 | 0101000000C00497D1162CB93F8CBAEF08A080E63F | Peter |  45 |    5.6 | 1965-04-12
     1 | 010100000054E943ACD697E2BFC0895EE54A46CF3F | Paul  |  33 |   5.84 | 1971-03-25
```

You can also apply filters, and see the portions that will be pushed down to the OGR driver, by setting the debug level to `DEBUG1`.

```sql
SET client_min_messages = debug1;

SELECT name, age, height
FROM pt_two
WHERE height < 5.7
AND geom && ST_MakeEnvelope(0, 0, 1, 1);
```

```
DEBUG:  OGR SQL: (height < 5.7)
DEBUG:  OGR spatial filter (0 0, 1 1)
 name  | age | height
-------+-----+--------
 Peter |  45 |    5.6
(1 row)
```

Examples

WFS FDW

Since we can access any OGR data source as a table, how about a public WFS server?

```sql
CREATE EXTENSION postgis;
CREATE EXTENSION ogr_fdw;
CREATE SERVER geoserver
    FOREIGN DATA WRAPPER ogr_fdw
    OPTIONS (
        datasource 'WFS:https://demo.geo-solutions.it/geoserver/wfs',
        format 'WFS' );

CREATE FOREIGN TABLE topp_states (
    fid bigint,
    the_geom Geometry(MultiSurface,4326),
    gml_id varchar,
    state_name varchar,
    state_fips varchar,
    sub_region varchar,
    state_abbr varchar,
    land_km double precision,
    water_km double precision,
    persons double precision,
    families double precision,
    houshold double precision,
    male double precision,
    female double precision,
    workers double precision,
    drvalone double precision,
    carpool double precision,
    pubtrans double precision,
    employed double precision,
    unemploy double precision,
    service double precision,
    manual double precision,
    p_male double precision,
    p_female double precision,
    samp_pop double precision
    )
    SERVER "geoserver"
    OPTIONS (layer 'topp:states');
```

FGDB FDW

Unzip the `Querying.zip` file from the `data` directory to get a `Querying.gdb` file, and put it somewhere public (like `/tmp`). Now run the `ogr_fdw_info` tool on it to get a table definition.

```
CREATE SERVER fgdbtest
    FOREIGN DATA WRAPPER ogr_fdw
    OPTIONS (
        datasource '/tmp/Querying.gdb',
        format 'OpenFileGDB' );

CREATE FOREIGN TABLE cities (
    fid integer,
    geom geometry(Point, 4326),
    city_fips varchar,
    city_name varchar,
    state_fips varchar,
    state_name varchar,
    state_city varchar,
    type varchar,
    capital varchar,
    elevation integer,
    pop1990 integer,
    popcat integer
    )
    SERVER fgdbtest
    OPTIONS (layer 'Cities');
```

Query away!

PostgreSQL FDW

Wraparound action! Handy for testing. Connect your database back to your database and watch the fur fly. This is only for testing, for best performance you should use postgres_fdw foreign data wrapper even when querying a PostGIS enabled database.

```
CREATE TABLE apostles (
    fid serial primary key,
    geom geometry(Point, 4326),
    joined integer,
    name text,
    height real,
    born date,
    clock time,
    ts timestamp
);

INSERT INTO apostles (name, geom, joined, height, born, clock, ts) VALUES
    ('Peter', 'SRID=4326;POINT(30.31 59.93)', 1, 1.6, '1912-01-10', '10:10:01', '1912-01-10
    ('Andrew', 'SRID=4326;POINT(-2.8 56.34)', 2, 1.8, '1911-02-11', '10:10:02', '1911-02-11
    ('James', 'SRID=4326;POINT(-79.23 42.1)', 3, 1.72, '1910-03-12', '10:10:03', '1910-03-12
    ('John', 'SRID=4326;POINT(13.2 47.35)', 4, 1.45, '1909-04-01', '10:10:04', '1909-04-01 1
    ('Philip', 'SRID=4326;POINT(-75.19 40.69)', 5, 1.65, '1908-05-02', '10:10:05', '1908-05-
    ('Bartholomew', 'SRID=4326;POINT(-62 18)', 6, 1.69, '1907-06-03', '10:10:06', '1907-06-0
    ('Thomas', 'SRID=4326;POINT(-80.08 35.88)', 7, 1.68, '1906-07-04', '10:10:07', '1906-07-
    ('Matthew', 'SRID=4326;POINT(-73.67 20.94)', 8, 1.65, '1905-08-05', '10:10:08', '1905-08
    ('James Alpheus', 'SRID=4326;POINT(-84.29 34.07)', 9, 1.78, '1904-09-06', '10:10:09', '1
    ('Thaddaeus', 'SRID=4326;POINT(79.13 10.78)', 10, 1.88, '1903-10-07', '10:10:10', '1903-
    ('Simon', 'SRID=4326;POINT(-85.97 41.75)', 11, 1.61, '1902-11-08', '10:10:11', '1902-11-
    ('Judas Iscariot', 'SRID=4326;POINT(35.7 32.4)', 12, 1.71, '1901-12-09', '10:10:12', '19

CREATE SERVER wraparound
    FOREIGN DATA WRAPPER ogr_fdw
    OPTIONS (
        datasource 'Pg:dbname=fdw user=postgres',
        format 'PostgreSQL' );

CREATE FOREIGN TABLE apostles_fdw (
    fid integer,
    geom geometry(Point, 4326),
    joined integer,
    name text,
    height real,
    born date,
    clock time,
    ts timestamp
)
SERVER wraparound
    OPTIONS (layer 'apostles');

SELECT * FROM apostles_fdw;
```

Advanced Features

Writeable FDW Tables

If the OGR driver you are using supports it, you can insert/update/delete records from your FDW tables.

For file-backed drivers, the user under which `postgres` runs will need read/write access to the file being altered. For database-backed drivers, your connection needs a user with read/write permissions to the database.

By default, servers and tables are updateable if the OGR driver supports it, but you can turn off updateability at a server or table level using the `updateable` option:

```
ALTER SERVER myserver
    OPTIONS (ADD updateable 'false');

ALTER FOREIGN TABLE mytable
    OPTIONS (ADD updateable 'false');
```

Writeable tables only work if you have included a `fid` column in your table definition. By default, tables imported by `IMPORT FOREIGN SCHEMA` or using the example SQL code from `ogr_fdw_info` include a `fid` column.

Column Name Mapping

You can create an FDW table with any subset of columns from the OGR source you like, just by using the same column names as the source:

```
CREATE FOREIGN TABLE typetest_fdw_partial (
    clock time,
    name varchar
    )
    SERVER wraparound
    OPTIONS (layer 'typetest');
```

You can also explicitly map remote column names to different local names using the `column_name` option:

```
CREATE FOREIGN TABLE typetest_fdw_mapped (
    fid bigint,
    supertime time OPTIONS (column_name 'clock'),
    thebestnamething varchar OPTIONS (column_name 'name')
    )
    SERVER wraparound
    OPTIONS (layer 'typetest');
```

Automatic Foreign Table Creation

This feature is only available with PostgreSQL 9.5 and higher

You can use the PostgreSQL `IMPORT FOREIGN SCHEMA` command to import table definitions from an OGR data source.

Import All Tables If you want to import all tables in the OGR data source use the special schema called "ogr_all".

```
CREATE SCHEMA fgdball;

IMPORT FOREIGN SCHEMA ogr_all
    FROM SERVER fgdbtest
    INTO fgdball;
```

Import a Subset of Tables Not all OGR data sources have a concept of schema, so we use the remote schema string as a prefix to match OGR layers. The matching is case sensitive, so make sure casing matches your layer names.

For example, the following will only import tables that start with *CitiesIn*. As long as you quote, you can handle true schemaed databases such as SQL Server or PostgreSQL by using something like *"dbo."*

```
CREATE SCHEMA fgdbcityinf;

IMPORT FOREIGN SCHEMA "CitiesIn"
    FROM SERVER fgdbtest
    INTO fgdbcityinf;
```

You can also use PostgreSQL clauses `LIMIT TO` and `EXCEPT` to restrict the tables you are importing.

```
CREATE SCHEMA fgdbcitysub;

-- import only layer called Cities
IMPORT FOREIGN SCHEMA ogr_all
    LIMIT TO(cities)
    FROM server fgdbtest
    INTO fgdbcitysub ;

-- import only layers not called Cities or Countries
IMPORT FOREIGN SCHEMA ogr_all
    EXCEPT (cities, countries)
    FROM server fgdbtest
    INTO fgdbcitysub;

-- With table laundering turned off, need to use exact layer names
DROP SCHEMA IF EXISTS fgdbcitysub CASCADE;

    -- import with un-laundered table name
IMPORT FOREIGN SCHEMA ogr_all
    LIMIT TO("Cities")
    FROM server fgdbtest
    INTO fgdbcitysub
    OPTIONS (launder_table_names 'false') ;
```

Mixed Case and Special Characters In general, PostgreSQL prefers table names with simple numbers and letters, no punctuation or special characters.

By default, when `IMPORT FOREIGN SCHEMA` is run on an OGR foreign data server, the table names and column names are "laundered" – all upper case is converted to lowercase and special characters such as spaces and punctuation are replaced with "_".

Laundering is not desirable in all cases. You can override this behavior with two `IMPORT FOREIGN SCHEMA` options specific to `ogr_fdw` servers: `launder_column_names` and `launder_table_names`.

To preserve casing and other funky characters in both column names and table names, do the following:

```
CREATE SCHEMA fgdbcitypreserve;

IMPORT FOREIGN SCHEMA ogr_all
    FROM SERVER fgdbtest
    INTO fgdbpreserve
    OPTIONS (
        launder_table_names 'false',
        launder_column_names 'false'
    );
```

GDAL Options

The behavior of your GDAL/OGR connection can be altered by passing GDAL `config_options` to the connection when you set up the server. Most GDAL/OGR drivers have some specific behaviours that are controlled by configuration options. For example, the "ESRI Shapefile" driver includes a `SHAPE_ENCODING` option that controls the character encoding applied to text data.

Since many Shapefiles are encoded using LATIN1, and most PostgreSQL databases are encoded in UTF-8, it is useful to specify the encoding to get proper handling of special characters like accents.

```
CREATE SERVER myserver_latin1
    FOREIGN DATA WRAPPER ogr_fdw
    OPTIONS (
        datasource '/tmp/test',
        format 'ESRI Shapefile',
        config_options 'SHAPE_ENCODING=LATIN1'
    );
```

Multiple config options can be passed at one time by supplying a **space-separated** list of options.

If you are using GDAL 2.0 or higher, you can also pass "open options" to your OGR foreign data wrapper, using the `open_options` parameter. In GDAL 2.0, the global `SHAPE_ENCODING` option has been superceded by a driver-specific `ENCODING` option, which can be called like this:

```
CREATE SERVER myserver_latin1
    FOREIGN DATA WRAPPER ogr_fdw
    OPTIONS (
        datasource '/tmp/test',
        format 'ESRI Shapefile',
        open_options 'ENCODING=LATIN1'
    );
```

GDAL Debugging

If you are getting odd behavior and you want to see what GDAL is doing behind the scenes, enable debug logging in your server:

```
CREATE SERVER myserver_latin1
    FOREIGN DATA WRAPPER ogr_fdw
    OPTIONS (
        datasource '/tmp/test',
        format 'ESRI Shapefile',
        config_options 'SHAPE_ENCODING=LATIN1 CPL_DEBUG=ON'
    );
```

GDAL-level messages will be logged at the PostgreSQL **DEBUG2** level, so to see them when running a query, alter your `client_min_messages` setting.

```
SET client_min_messages = debug2;
```

Once you've figured out your issue, don't forget to remove the `CPL_DEBUG` option from your server definition, and set your messages back to **NOTICE** level.

```
SET client_min_messages = notice;
ALTER SERVER myserver_latin1
    OPTIONS (
        SET config_options 'SHAPE_ENCODING=LATIN1'
    );
```

Utility Functions

To view the current FDW and GDAL version.

```
SELECT ogr_fdw_version();
```

To view the drivers supported by this GDAL.

```
SELECT unnest(ogr_fdw_drivers());
```

Character Encoding

To access sources that have a non-UTF-8 encoding, you may need to specify the character encoding in your server creation line. OGR FDW uses the transcoding built into PostgreSQL, and thus supports all the encodings that PostgreSQL does.

```
CREATE SERVER odbc_latin1
    FOREIGN DATA WRAPPER ogr_fdw
    OPTIONS (
        datasource 'ODBC:username@servicename',
        format 'ODBC',
        character_encoding 'WIN1250'
    );
```

```
CREATE FOREIGN TABLE featuretable_fdw (
    name text,
    geom geometry(Point, 4326)
)
SERVER odbc_latin1
    OPTIONS (layer 'featuretable');
```

Frequently Asked Questions

ODBC connections work in `ogr_fdw_info.exe` but not in the database, why?

1. One possibility is that you registered DSN under USER instead of System. It should be system since otherwise it will only work under the account you are logged in as. Since `ogr_fdw_info.exe` is a client app, it will give you a false sense of success since ODBC will in server, run under the context of the PostgreSQL service account. Verify that you have a System DSN (and **not** a User DSN).
2. If you used the Windows installer for PostgreSQL, it starts up PostgreSQL using Network Service account. I always manually switch it to a real user account since I need it to access some network resources. I never tested, but I suspect ODBC keys may not be readable by Network Service account.

If change #1 doesn't work, try changing PostgreSQL to run under a regular user account. Make sure that user has full control of the PostgreSQL data folder.

Do I need ODBC to read an MS Access database?

You shouldn't need to do ODBC for MS Access, should be able to do:

```
ogr_fdw_info.exe -s "D:\FDW Data\My Folder\MYFILE.MDB"
```

As an added bonus, using a direct access shouldn't need to read ODBC registry keys since it's a DNSless connection.

Why doesn't my MS Access connection work?

If your MS Access database is on a network share, your PostgreSQL service account needs to be able to access it by the path you use. For MS Access databases it also has to have write permissions. The reason is MS Access databases use a locking file to manage access, so all users that have read permission also need to have write permission into the folder to create the lock file and delete the lock file (if they are the last ones in) if it doesn't exist.

Also note: even if the PostgreSQL service account can access the folder `\\S\Files`, if you map it to say `S:\` the connection will not work if the mapped drive is not set under the user account that PostgreSQL runs under. (That said – it's safer to use UNCs (`\\S\Files`) instead of mapped drives.