file_fdw: Directly Query Flat Files in PostgreSQL Without Importing

The file_fdw (Foreign Data Wrapper) is a PostgreSQL extension that lets you access data stored in flat files, like CSV files, as if they were regular tables in your PostgreSQL database. This is useful for integrating external data sources without needing to import the data directly into your database.
file_fdw is a contrib module, meaning it’s an additional feature included with PostgreSQL but not part of its core functionality. Contrib modules provide extra capabilities and enhancements beyond the core database system.

- It allows you to integrate and query data from flat files directly within PostgreSQL, making it easier to combine and analyze data from different sources.
- By treating flat files as tables, you can simplify Extract, Transform, Load (ETL) processes, making it easier to load data into your database.
- You can perform ad-hoc queries on external data without needing to import it into your database, which can save time and storage space.

How to use file_fdw

To use file_fdw we must create its extension

```
CREATE EXTENSION file_fdw;Copy to Clipboard
```

Once the extension is created we then need to create a foreign table
Create foreign table

```
CREATE FOREIGN TABLE readings (

    tags_id integer[],

    time timestamp with time zone[],

    latitude double precision[],

    longitude double precision[],

    elevation double precision[],

    velocity double precision[],

    heading double precision[],

    grade double precision[],

    fuel_consumption double precision[]

) SERVER file_server

OPTIONS (

    filename '<PATH_TO_LOCAL_CSV_FILE>',

    format 'csv',

    header 'true'

);Copy to Clipboard
```

Where **<PATH_TO_LOCAL_CSV_FILE>** represents the actual location of the data file on your system. For example, in my case, it was **/home/ubuntu/data.csv**.
We can now simply do

```
postgres=# select * from my_file_table limit 1;

-[ RECORD 1 ]----+--------------------------------------------------------------------------
--------------------------------------------

tags_id          |
{3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
,3,3,3}

time             | {"2024-01-01 22:15:10+00","2024-01-01 21:56:40+00","2024-01-01
21:46:00+00","2024-01-01 22:18:50+00","2024-01-01 21:21:10+00","2024-01-01 21:14:40+00","2024-
01-01 21:19:00+00","2024-01-01 21:54:10+00","2024-01-01 21:22:50+00","2024-01-01
21:20:20+00","2024-01-01 21:15:10+00","2024-01-01 21:52:00+00","2024-01-01 21:17:00+00","2024-
01-01 21:59:30+00","2024-01-01 22:20:40+00","2024-01-01 21:39:40+00","2024-01-01
21:19:40+00","2024-01-01 21:15:50+00","2024-01-01 21:17:10+00","2024-01-01 22:20:50+00","2024-
01-01 21:53:10+00","2024-01-01 22:12:30+00","2024-01-01 21:18:50+00","2024-01-01
21:59:50+00","2024-01-01 21:48:00+00","2024-01-01 21:45:20+00","2024-01-01 21:42:20+00","2024-
01-01 21:42:00+00","2024-01-01 21:20:10+00","2024-01-01 21:44:50+00","2024-01-01
21:45:10+00","2024-01-01 22:15:00+00","2024-01-01 21:56:30+00","2024-01-01 21:53:20+00","2024-
01-01 22:18:20+00","2024-01-01 22:13:50+00","2024-01-01 22:20:30+00","2024-01-01
22:19:10+00","2024-01-01 21:43:50+00","2024-01-01 22:22:10+00","2024-01-01 21:51:20+00","2024-
01-01 21:19:20+00","2024-01-01 22:14:00+00","2024-01-01 21:40:20+00","2024-01-01
21:18:40+00","2024-01-01 21:16:30+00","2024-01-01 23:58:50+00","2024-01-01 23:52:50+00","2024-
01-01 23:52:30+00","2024-01-01 23:34:00+00"}

latitude         |
{82.11994,82.08518,82.09893,82.11312,82.09393,82.07374,82.08686,82.07246,82.08858,82.09129,82.
07738,82.06916,82.0758,82.09786,82.09837,82.11116,82.09256,82.07307,82.07205,82.09983,82.06623
,82.10301,82.0827,82.10537,82.09597,82.10508,82.10948,82.10801,82.09035,82.10564,82.10687,82.1
2051,82.08157,82.06952,82.12489,82.1109,82.09951,82.10851,82.10127,82.10305,82.07741,82.08822,
82.11187,82.11675,82.08066,82.08002,82.15954,82.16258,82.17037,82.16023}

longitude        |
{56.00129,55.97968,55.99326,56.0107,55.93018,55.90754,55.91676,55.98281,55.94279,55.93265,55.9
1122,55.97472,55.91691,55.96889,56.01456,55.99963,55.92866,55.91526,55.9132,56.01645,55.9811,5
5.98026,55.91561,55.9733,55.97965,55.9921,55.99804,56.00134,55.93315,NULL,55.99645,55.99813,55
.98306,55.98028,56.01018,55.98644,56.01231,56.01544,56.00471,56.012,55.96815,55.9232,55.99034,
56.00014,55.91408,55.91746,56.10255,56.10072,56.10132,56.10508}

elevation        |
{147,82,50,108,163,147,126,60,139,152,138,44,132,NULL,128,53,134,117,123,119,70,135,132,70,37,
46,17,27,149,45,38,148,84,63,116,161,129,108,13,142,23,136,159,45,134,127,121,123,126,133}

velocity         |
{76,71,53,63,58,89,47,53,51,39,86,75,58,48,71,0,33,79,66,72,65,56,46,55,45,52,40,43,38,34,43,6
7,67,58,61,73,75,74,32,58,74,36,72,4,53,75,3,44,38,86}

heading          |
{79,56,45,69,18,53,41,61,25,31,59,59,47,61,68,58,38,53,42,64,60,71,40,60,50,46,48,54,30,49,45,
84,53,61,72,78,67,68,41,58,58,46,81,NULL,42,53,30,26,21,0}

grade            |
{88,72,92,87,100,94,96,76,95,96,92,61,100,74,78,95,95,100,97,78,62,92,92,84,72,89,99,98,NULL,7
9,87,86,68,62,88,93,80,83,87,70,59,96,89,100,94,99,17,18,16,17}

fuel_consumption |
{39.1,31.3,47.7,48.7,0,12.1,5.1,29.6,0,0,16.8,48.3,25.1,15.5,32.7,28.9,4,25,20.9,35.2,40.4,42.
8,7,20.3,41.9,43.4,42.3,38,0,50,44.4,40,34,37,48.8,35.3,37.3,46.9,50,49.1,48.1,5.4,37.3,32.9,4
.4,26.8,12.8,13.5,10.8,16.5}Copy to Clipboard
```

Since the data in my original CSV file was organized as column arrays, we can use
the **UNNEST** function to convert it into simple rows in PostgreSQL. To accomplish this, I will create a new PostgreSQL table named **normalized_readings**.

```
CREATE TABLE normalized_readings (tags_id integer, time timestamp with time zone, latitude
double precision, longitude double precision, elevation double precision, velocity double
precision, heading double precision, grade double precision, fuel_consumption double
precision);Copy to Clipboard
```

Now, we can use **INSERT** along with **UNNEST** to both insert data and transform column arrays into simple rows.

```
INSERT INTO normalized_readings (tags_id, time, latitude, longitude, elevation, velocity,
heading, grade, fuel_consumption)

SELECT

    unnest(tags_id) AS tags_id,

    unnest(time) AS time,

    unnest(latitude) AS latitude,

    unnest(longitude) AS longitude,

    unnest(elevation) AS elevation,

    unnest(velocity) AS velocity,

    unnest(heading) AS heading,

    unnest(grade) AS grade,

    unnest(fuel_consumption) AS fuel_consumption

FROM

    readings;Copy to Clipboard
```

Now, if I query the table, the results will be returned in row format.

```
postgres=# select * from normalized_readings limit 1;

 tags_id |          time          | latitude | longitude | elevation | velocity | heading |
grade | fuel_consumption

---------+------------------------+----------+-----------+-----------+----------+---------+---
----+------------------

       3 | 2024-01-01 22:15:10+00 | 82.11994 |  56.00129 |       147 |       76 |      79
|    88 |             39.1

(1 row)Copy to Clipboard
```

We have successfully shown how to use file_fdw to read data directly from a CSV file and how to manipulate the data according to our needs.
file_fdw can be a valuable addition to your PostgreSQL toolkit, simplifying data access and analysis in ways that traditional data import methods might not.