F.35. postgres_fdw

The `postgres_fdw` module provides the foreign-data wrapper `postgres_fdw`, which can be used to access data stored in external PostgreSQL servers.

The functionality provided by this module overlaps substantially with the functionality of the older **dblink** module. But `postgres_fdw` provides more transparent and standards-compliant syntax for accessing remote tables, and can give better performance in many cases.

To prepare for remote access using `postgres_fdw`:

1. Install the `postgres_fdw` extension using **CREATE EXTENSION**.

2. Create a foreign server object, using **CREATE SERVER**, to represent each remote database you want to connect to. Specify connection information, except `user` and `password`, as options of the server object.

3. Create a user mapping, using CREATE USER MAPPING, for each database user you want to allow to access each foreign server. Specify the remote user name and password to use as `user` and `password` options of the user mapping.

4. Create a foreign table, using CREATE FOREIGN TABLE or IMPORT FOREIGN SCHEMA, for each remote table you want to access. The columns of the foreign table must match the referenced remote table. You can, however, use table and/or column names different from the remote table's, if you specify the correct remote names as options of the foreign table object.

Now you need only `SELECT` from a foreign table to access the data stored in its underlying remote table. You can also modify the remote table using `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE`. (Of course, the remote user you have specified in your user mapping must have privileges to do these things.)

Note that the `ONLY` option specified in `SELECT`, `UPDATE`, `DELETE` or `TRUNCATE` has no effect when accessing or modifying the remote table.

Note that `postgres_fdw` currently lacks support for `INSERT` statements with an `ON CONFLICT DO UPDATE` clause. However, the `ON CONFLICT DO NOTHING` clause is supported, provided a unique index inference specification is omitted. Note also that `postgres_fdw` supports row movement invoked by `UPDATE` statements executed on partitioned tables, but it currently does not handle the case where a remote partition chosen to insert a moved row into is also an `UPDATE` target partition that will be updated elsewhere in the same command.

It is generally recommended that the columns of a foreign table be declared with exactly the same data types, and collations if applicable, as the referenced columns of the remote table. Although `postgres_fdw` is currently rather forgiving about performing data type conversions at need, surprising semantic anomalies may arise when types or collations do not match, due to the remote server interpreting query conditions differently from the local server.

Note that a foreign table can be declared with fewer columns, or with a different column order, than its underlying remote table has. Matching of columns to the remote table is by name, not position.

F.35.1. FDW Options of postgres_fdw

F.35.1.1. Connection Options

A foreign server using the `postgres_fdw` foreign data wrapper can have the same options that libpq accepts in connection strings, as described in Section 34.1.2, except that these options are not allowed or have special handling:

- `user`, `password` and `sslpassword` (specify these in a user mapping, instead, or use a service file)
- `client_encoding` (this is automatically set from the local server encoding)
- `fallback_application_name` (always set to `postgres_fdw`)

- `sslkey` and `sslcert` - these may appear in *either or both* a connection and a user mapping. If both are present, the user mapping setting overrides the connection setting.

Only superusers may create or modify user mappings with the `sslcert` or `sslkey` settings.

Only superusers may connect to foreign servers without password authentication, so always specify the `password` option for user mappings belonging to non-superusers.

A superuser may override this check on a per-user-mapping basis by setting the user mapping option `password_required 'false'`, e.g.,

```
ALTER USER MAPPING FOR some_non_superuser SERVER loopback_nopw
OPTIONS (ADD password_required 'false');
```

To prevent unprivileged users from exploiting the authentication rights of the unix user the postgres server is running as to escalate to superuser rights, only the superuser may set this option on a user mapping.

Care is required to ensure that this does not allow the mapped user the ability to connect as superuser to the mapped database per CVE-2007-3278 and CVE-2007-6601. Don't set `password_required=false` on the `public` role. Keep in mind that the mapped user can potentially use any client certificates, `.pgpass`, `.pg_service.conf` etc in the unix home directory of the system user the postgres server runs as. They can also use any trust relationship granted by authentication modes like `peer` or `ident` authentication.

F.35.1.2. Object Name Options

These options can be used to control the names used in SQL statements sent to the remote PostgreSQL server. These options are needed when a foreign table is created with names different from the underlying remote table's names.

`schema_name`

This option, which can be specified for a foreign table, gives the schema name to use for the foreign table on the remote server. If this option is omitted, the name of the foreign table's schema is used.

`table_name`

This option, which can be specified for a foreign table, gives the table name to use for the foreign table on the remote server. If this option is omitted, the foreign table's name is used.

`column_name`

This option, which can be specified for a column of a foreign table, gives the column name to use for the column on the remote server. If this option is omitted, the column's name is used.

F.35.1.3. Cost Estimation Options

`postgres_fdw` retrieves remote data by executing queries against remote servers, so ideally the estimated cost of scanning a foreign table should be whatever it costs to be done on the remote server, plus some overhead for communication. The most reliable way to get such an estimate is to ask the remote server and then add something for overhead — but for simple queries, it may not be worth the cost of an additional remote query to get a cost estimate. So `postgres_fdw` provides the following options to control how cost estimation is done:

`use_remote_estimate`

This option, which can be specified for a foreign table or a foreign server, controls whether `postgres_fdw` issues remote `EXPLAIN` commands to obtain cost estimates. A setting for a foreign table overrides any setting for its server, but only for that table. The default is `false`.

`fdw_startup_cost`

This option, which can be specified for a foreign server, is a floating point value that is added to the estimated startup cost of any foreign-table scan on that server. This represents the additional overhead of establishing a connection, parsing and planning the query on the remote side, etc. The default value is `100`.

`fdw_tuple_cost`

This option, which can be specified for a foreign server, is a floating point value that is used as extra cost per-tuple for foreign-table scans on that server. This represents the additional overhead of data transfer between servers. You might increase or decrease this number to reflect higher or lower network delay to the remote server. The default value is `0.01`.

When `use_remote_estimate` is true, `postgres_fdw` obtains row count and cost estimates from the remote server and then adds `fdw_startup_cost` and `fdw_tuple_cost` to the cost estimates. When `use_remote_estimate` is false, `postgres_fdw` performs local row count and cost estimation and then adds `fdw_startup_cost` and `fdw_tuple_cost` to the cost estimates. This local estimation is unlikely to be very accurate unless local copies of the remote table's statistics are available. Running ANALYZE on the foreign table is the way to update the local statistics; this will perform a scan of the remote table and then calculate and store statistics just as though the table were local. Keeping local statistics can be a useful way to reduce per-query planning overhead for a remote table — but if the remote table is frequently updated, the local statistics will soon be obsolete.

F.35.1.4. Remote Execution Options

By default, only `WHERE` clauses using built-in operators and functions will be considered for execution on the remote server. Clauses involving non-built-in functions are checked locally after rows are fetched. If such functions are available on the remote server and can be relied on to produce the same results as they do locally, performance can be improved by sending such `WHERE` clauses for remote execution. This behavior can be controlled using the following option:

`extensions`

This option is a comma-separated list of names of PostgreSQL extensions that are installed, in compatible versions, on both the local and remote servers. Functions and operators that are immutable and belong to a listed extension will be considered shippable to the remote server. This option can only be specified for foreign servers, not per-table.

When using the `extensions` option, *it is the user's responsibility* that the listed extensions exist and behave identically on both the local and remote servers. Otherwise, remote queries may fail or behave unexpectedly.

`fetch_size`

This option specifies the number of rows `postgres_fdw` should get in each fetch operation. It can be specified for a foreign table or a foreign server. The option specified on a table overrides an option specified for the server. The default is `100`.

`batch_size`

This option specifies the number of rows `postgres_fdw` should insert in each insert operation. It can be specified for a foreign table or a foreign server. The option specified on a table overrides an option specified for the server. The default is `1`.

Note the actual number of rows `postgres_fdw` inserts at once depends on the number of columns and the provided `batch_size` value. The batch is executed as a single query, and the libpq protocol (which `postgres_fdw` uses to connect to a remote server) limits the number of parameters in a single query to 65535. When the number of columns * `batch_size` exceeds the limit, the `batch_size` will be adjusted to avoid an error.

F.35.1.5. Asynchronous Execution Options

`postgres_fdw` supports asynchronous execution, which runs multiple parts of an `Append` node concurrently rather than serially to improve performance. This execution can be controlled using the following option:

`async_capable`

This option controls whether `postgres_fdw` allows foreign tables to be scanned concurrently for asynchronous execution. It can be specified for a foreign table or a foreign server. A table-level option overrides a server-level option. The default is `false`.

In order to ensure that the data being returned from a foreign server is consistent, `postgres_fdw` will only open one connection for a given foreign server and will run all queries against that server sequentially even if there are multiple foreign tables involved, unless those tables are subject to different user mappings. In such a case, it may be more performant to disable this option to eliminate the overhead associated with running queries asynchronously.

Asynchronous execution is applied even when an `Append` node contains subplan(s) executed synchronously as well as subplan(s) executed asynchronously. In such a case, if the asynchronous subplans are ones processed using `postgres_fdw`, tuples from the asynchronous subplans are not returned until after at least one synchronous subplan returns all tuples, as that subplan is executed while the asynchronous subplans are waiting for the results of asynchronous queries sent to foreign servers. This behavior might change in a future release.

F.35.1.6. Updatability Options

By default all foreign tables using `postgres_fdw` are assumed to be updatable. This may be overridden using the following option:

`updatable`

This option controls whether `postgres_fdw` allows foreign tables to be modified using `INSERT`, `UPDATE` and `DELETE` commands. It can be specified for a foreign table or a foreign server. A table-level option overrides a server-level option. The default is `true`.

Of course, if the remote table is not in fact updatable, an error would occur anyway. Use of this option primarily allows the error to be thrown locally without querying the remote server. Note however that the `information_schema` views will report a `postgres_fdw` foreign table to be updatable (or not) according to the setting of this option, without any check of the remote server.

F.35.1.7. Truncatability Options

By default all foreign tables using `postgres_fdw` are assumed to be truncatable. This may be overridden using the following option:

`truncatable`

This option controls whether `postgres_fdw` allows foreign tables to be truncated using the `TRUNCATE` command. It can be specified for a foreign table or a foreign server. A table-level option overrides a server-level option. The default is `true`.

Of course, if the remote table is not in fact truncatable, an error would occur anyway. Use of this option primarily allows the error to be thrown locally without querying the remote server.

F.35.1.8. Importing Options

`postgres_fdw` is able to import foreign table definitions using IMPORT FOREIGN SCHEMA. This command creates foreign table definitions on the local server that match tables or views present on the remote server. If the remote tables to be imported have columns of user-defined data types, the local server must have compatible types of the same names.

Importing behavior can be customized with the following options (given in the `IMPORT FOREIGN SCHEMA` command):

`import_collate`

> This option controls whether column `COLLATE` options are included in the definitions of foreign tables imported from a foreign server. The default is `true`. You might need to turn this off if the remote server has a different set of collation names than the local server does, which is likely to be the case if it's running on a different operating system. If you do so, however, there is a very severe risk that the imported table columns' collations will not match the underlying data, resulting in anomalous query behavior.
>
> Even when this parameter is set to `true`, importing columns whose collation is the remote server's default can be risky. They will be imported with `COLLATE "default"`, which will select the local server's default collation, which could be different.

`import_default`

> This option controls whether column `DEFAULT` expressions are included in the definitions of foreign tables imported from a foreign server. The default is `false`. If you enable this option, be wary of defaults that might get computed differently on the local server than they would be on the remote server; `nextval()` is a common source of problems. The `IMPORT` will fail altogether if an imported default expression uses a function or operator that does not exist locally.

`import_generated`

> This option controls whether column `GENERATED` expressions are included in the definitions of foreign tables imported from a foreign server. The default is `true`. The `IMPORT` will fail altogether if an imported generated expression uses a function or operator that does not exist locally.

`import_not_null`

> This option controls whether column `NOT NULL` constraints are included in the definitions of foreign tables imported from a foreign server. The default is `true`.

Note that constraints other than `NOT NULL` will never be imported from the remote tables. Although PostgreSQL does support check constraints on foreign tables, there is no provision for importing them automatically, because of the risk that a constraint expression could evaluate differently on the local and remote servers. Any such inconsistency in the behavior of a check constraint could lead to hard-to-detect errors in

query optimization. So if you wish to import check constraints, you must do so manually, and you should verify the semantics of each one carefully. For more detail about the treatment of check constraints on foreign tables, see CREATE FOREIGN TABLE.

Tables or foreign tables which are partitions of some other table are imported only when they are explicitly specified in `LIMIT TO` clause. Otherwise they are automatically excluded from IMPORT FOREIGN SCHEMA. Since all data can be accessed through the partitioned table which is the root of the partitioning hierarchy, importing only partitioned tables should allow access to all the data without creating extra objects.

F.35.1.9. Connection Management Options

By default, all connections that `postgres_fdw` establishes to foreign servers are kept open in the local session for re-use.

`keep_connections`

This option controls whether `postgres_fdw` keeps the connections to the foreign server open so that subsequent queries can re-use them. It can only be specified for a foreign server. The default is `on`. If set to `off`, all connections to this foreign server will be discarded at the end of each transaction.

F.35.2. Functions

`postgres_fdw_get_connections(OUT server_name text, OUT valid boolean) returns setof record`

This function returns the foreign server names of all the open connections that `postgres_fdw` established from the local session to the foreign servers. It also returns whether each connection is valid or not. `false` is returned if the foreign server connection is used in the current local transaction but its foreign server or user mapping is changed or dropped (Note that server name of an invalid connection will be `NULL` if the server is dropped), and then such invalid connection will be closed at the end of that transaction. `true` is returned otherwise. If there are no open connections, no record is returned. Example usage of the function:

```
postgres=# SELECT * FROM postgres_fdw_get_connections() ORDER BY 1;
 server_name | valid
-------------+-------
 loopback1   | t
 loopback2   | f
```

`postgres_fdw_disconnect(server_name text) returns boolean`

This function discards the open connections that are established by `postgres_fdw` from the local session to the foreign server with the given name. Note that there can be multiple connections to the given server using different user mappings. If the connections are used in the current local transaction, they are not disconnected and warning messages are reported. This function returns `true` if it disconnects at least one connection, otherwise `false`. If no foreign server with the given name is found, an error is reported. Example usage of the function:

```
postgres=# SELECT postgres_fdw_disconnect('loopback1');
 postgres_fdw_disconnect
-------------------------
 t
```

`postgres_fdw_disconnect_all() returns boolean`

This function discards all the open connections that are established by `postgres_fdw` from the local session to foreign servers. If the connections are used in the current local transaction, they are not disconnected and warning messages are reported. This function returns `true` if it disconnects at least one connection, otherwise `false`. Example usage of the function:

```
postgres=# SELECT postgres_fdw_disconnect_all();
 postgres_fdw_disconnect_all
-----------------------------
 t
```

F.35.3. Connection Management

`postgres_fdw` establishes a connection to a foreign server during the first query that uses a foreign table associated with the foreign server. By default this connection is kept and re-used for subsequent queries in the same session. This behavior can be controlled using `keep_connections` option for a foreign server. If multiple user identities (user mappings) are used to access the foreign server, a connection is established for each user mapping.

When changing the definition of or removing a foreign server or a user mapping, the associated connections are closed. But note that if any connections are in use in the current local transaction, they are kept until the end of the transaction. Closed connections will be re-established when they are necessary by future queries using a foreign table.

Once a connection to a foreign server has been established, it's by default kept until the local or corresponding remote session exits. To disconnect a connection explicitly, `keep_connections` option for a foreign server may be disabled, or `postgres_fdw_disconnect` and `postgres_fdw_disconnect_all` functions may be used. For example, these are useful to close connections that are no longer necessary, thereby releasing connections on the foreign server.

F.35.4. Transaction Management

During a query that references any remote tables on a foreign server, `postgres_fdw` opens a transaction on the remote server if one is not already open corresponding to the current local transaction. The remote transaction is committed or aborted when the local transaction commits or aborts. Savepoints are similarly managed by creating corresponding remote savepoints.

The remote transaction uses `SERIALIZABLE` isolation level when the local transaction has `SERIALIZABLE` isolation level; otherwise it uses `REPEATABLE READ` isolation level. This choice ensures that if a query performs multiple table scans on the remote server, it will get snapshot-consistent results for all the scans. A consequence is that successive queries within a single transaction will see the same data from the remote server, even if concurrent updates are occurring on the remote server due to other activities. That behavior would be expected anyway if the local transaction uses `SERIALIZABLE` or `REPEATABLE READ` isolation level, but it might be surprising for a `READ COMMITTED` local transaction. A future PostgreSQL release might modify these rules.

Note that it is currently not supported by `postgres_fdw` to prepare the remote transaction for two-phase commit.

F.35.5. Remote Query Optimization

`postgres_fdw` attempts to optimize remote queries to reduce the amount of data transferred from foreign servers. This is done by sending query `WHERE` clauses to the remote server for execution, and by not retrieving table columns that are not needed for the current query. To reduce the risk of misexecution of queries, `WHERE` clauses are not sent to the remote server unless they use only data types, operators, and functions that are built-in or belong to an extension that's listed in the foreign server's `extensions` option. Operators and functions in such clauses must be `IMMUTABLE` as well. For an `UPDATE` or `DELETE` query, `postgres_fdw` attempts to optimize the query execution by sending the whole query to the remote server if there are no query `WHERE` clauses that cannot be sent to the remote server, no local joins for the query, no row-level local `BEFORE` or `AFTER` triggers or stored generated columns on the target table, and no `CHECK OPTION` constraints from parent views. In `UPDATE`, expressions to assign to target columns must use only built-in data types, `IMMUTABLE` operators, or `IMMUTABLE` functions, to reduce the risk of misexecution of the query.

When `postgres_fdw` encounters a join between foreign tables on the same foreign server, it sends the entire join to the foreign server, unless for some reason it believes that it will be more efficient to fetch rows from each table individually, or unless the table references involved are subject to different user mappings. While sending the `JOIN` clauses, it takes the same precautions as mentioned above for the `WHERE` clauses.

The query that is actually sent to the remote server for execution can be examined using `EXPLAIN VERBOSE`.

F.35.6. Remote Query Execution Environment

In the remote sessions opened by `postgres_fdw`, the search_path parameter is set to just `pg_catalog`, so that only built-in objects are visible without schema qualification. This is not an issue for queries generated by `postgres_fdw` itself, because it always supplies such qualification. However, this can pose a hazard for functions that are executed on the remote server via triggers or rules on remote tables. For example, if a remote table is actually a view, any functions used in that view will be executed with the restricted search path. It is recommended to schema-qualify all names in such functions, or else attach `SET search_path` options (see CREATE FUNCTION) to such functions to establish their expected search path environment.

`postgres_fdw` likewise establishes remote session settings for various parameters:

- TimeZone is set to `UTC`
- DateStyle is set to `ISO`
- IntervalStyle is set to `postgres`
- extra_float_digits is set to `3` for remote servers 9.0 and newer and is set to `2` for older versions

These are less likely to be problematic than `search_path`, but can be handled with function `SET` options if the need arises.

It is *not* recommended that you override this behavior by changing the session-level settings of these parameters; that is likely to cause `postgres_fdw` to malfunction.

F.35.7. Cross-Version Compatibility

`postgres_fdw` can be used with remote servers dating back to PostgreSQL 8.3. Read-only capability is available back to 8.1. A limitation however is that `postgres_fdw` generally assumes that immutable built-in functions and operators are safe to send to the remote server for execution, if they appear in a `WHERE` clause for a foreign table. Thus, a built-in function that was added since the remote server's release might be sent to it for execution, resulting in “function does not exist” or a similar error. This type of failure can be worked around by rewriting the query, for example by embedding the foreign table reference in a sub-`SELECT` with `OFFSET 0` as an optimization fence, and placing the problematic function or operator outside the sub-`SELECT`.

F.35.8. Examples

Here is an example of creating a foreign table with `postgres_fdw`. First install the extension:

```
CREATE EXTENSION postgres_fdw;
```

Then create a foreign server using **CREATE SERVER**. In this example we wish to connect to a PostgreSQL server on host `192.83.123.89` listening on port `5432`. The database to which the connection is made is named `foreign_db` on the remote server:

```
CREATE SERVER foreign_server
        FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host '192.83.123.89', port '5432', dbname 'foreign_db');
```

A user mapping, defined with **CREATE USER MAPPING**, is needed as well to identify the role that will be used on the remote server:

```
CREATE USER MAPPING FOR local_user
        SERVER foreign_server
        OPTIONS (user 'foreign_user', password 'password');
```

Now it is possible to create a foreign table with **CREATE FOREIGN TABLE**. In this example we wish to access the table named `some_schema.some_table` on the remote server. The local name for it will be `foreign_table`:

```
CREATE FOREIGN TABLE foreign_table (
        id integer NOT NULL,
        data text
)
        SERVER foreign_server
        OPTIONS (schema_name 'some_schema', table_name 'some_table');
```

It's essential that the data types and other properties of the columns declared in `CREATE FOREIGN TABLE` match the actual remote table. Column names must match as well, unless you attach `column_name` options to the individual columns to show how they are named in the remote table. In many cases, use of `IMPORT FOREIGN SCHEMA` is preferable to constructing foreign table definitions manually.

F.35.9. Author

Shigeru Hanada <shigeru.hanada@gmail.com>

Submit correction

If you see anything in the documentation that is not correct, does not match your experience with the particular feature or requires further clarification, please use this form to report a documentation issue.