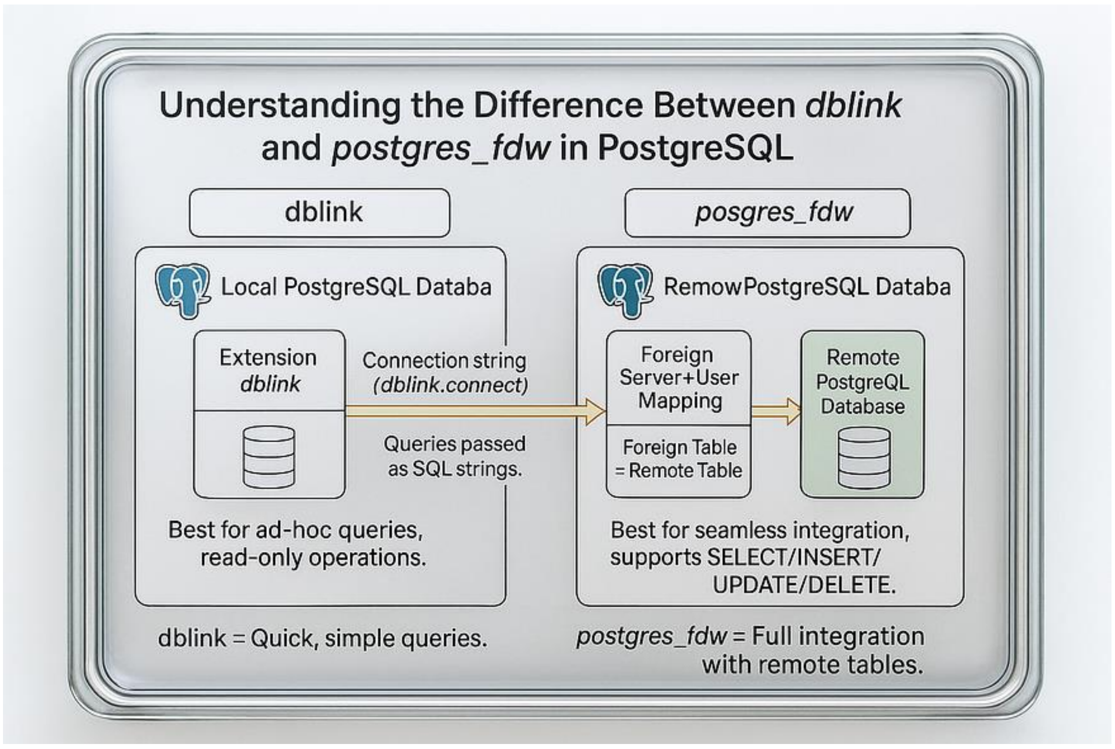
Understanding the Difference Between `dblink` and `postgres_fdw` i n PostgreSQL



Understanding the Difference Between *dblink* and *postgres_fdw* in PostgreSQL
dblink
posgres_fdw
Local PostgreSQL Databa
RemowPostgreSQL Databa
Extension *dblink*
Connection string (*dblink.connect*)
Queries passed as SQL strings.
Foreign Server+User Mapping
Foreign Table = Remote Table
Remote PostgreQL Database
Best for ad-hoc queries, read-only operations.
Best for seamless integration, supports SELECT/INSERT/ UPDATE/DELETE.
dblink = Quick, simple queries.
postgres_fdw = Full integration with remote tables.

PostgreSQL is not just a relational database; it is a highly extensible system. Two of its most commonly used extensions for working across multiple PostgreSQL databases are `dblink` and `postgres_fdw` **(Foreign Data Wrapper)**.

At first glance, they might look similar because both allow you to connect to another PostgreSQL database. But in reality, they serve different purposes and have different strengths.

□ 1. What is `dblink`?

The `dblink` **extension** is one of the oldest ways to connect one PostgreSQL database to another. It allows you to run **queries as text strings** against a remote database.

- Think of it as a **bridge**: you connect to the other database, send a query as text, and get the results back.
- You must manually define the expected column structure when pulling data.
- It's lightweight but also limited — best suited for **ad-hoc queries**.

Example workflow with `dblink`:

1. Install and enable the extension (`CREATE EXTENSION dblink`).
2. Configure a remote connection.
3. Run a query like:

```
SELECT *
FROM dblink('conn_db_link', 'SELECT item_id, quantity, price FROM items')
AS result(item_id INT, quantity INT, price NUMERIC(10,2));



postgres=# SELECT *
postgres-# FROM dblink('conn_db_link', 'SELECT item_id, quantity, price FROM items')
postgres-# AS result(item_id INT, quantity INT, price NUMERIC(10,2));
 item_id  | quantity | price
----------+----------+--------
  9093121 |      297 | 395.80
  9093122 |      640 | 226.10
  9093123 |      238 | 173.32
  9093124 |      200 |  80.91
  9093125 |       18 | 222.13
  9093126 |      945 | 101.12
  9093127 |      419 | 192.06
  9093128 |      739 | 493.85
  9093129 |      755 | 163.90
  9093130 |      374 |  69.76
  9093131 |      613 | 339.56
  9093132 |      923 |  75.71
  9093133 |      917 | 461.97
  9093134 |      916 | 156.12
  9093135 |      719 | 433.66
```

Here, the SQL query is written as a string and executed on the remote server.

2. What is `postgres_fdw`?

The `postgres_fdw` **extension** is a modern, more powerful replacement for many use cases of `dblink`. FDW stands for **Foreign Data Wrapper**, which is part of PostgreSQL's implementation of the SQL/MED standard (Management of External Data).

- Instead of sending queries as strings, you can create **foreign tables** in your local database that directly map to remote tables.
- These foreign tables look and behave like normal local tables.
- You can **SELECT, INSERT, UPDATE, and DELETE** records across databases.
- PostgreSQL's **query planner** can optimize queries across local and remote data sources.

Example workflow with `postgres_fdw`:

1. Install and enable the extension (`CREATE EXTENSION postgres_fdw`).
2. Create a foreign server that points to the remote database.
3. Create a user mapping with credentials.
4. Define a foreign table that mirrors the remote table.
5. Run queries against it directly, without writing raw SQL strings.

```sql
SELECT * FROM items_remote WHERE price > 100;
```

The query above looks like a local query, but it is executed on the remote database behind the scenes.

□ 3. Key Differences at a Glance

Feature `dblink` `postgres_fdw` **Query style** SQL passed as a string Normal SQL on foreign tables **Integration level** Lightweight, ad-hoc Full integration with local schema **Operations supported** Mostly SELECT (writes are clunky) SELECT, INSERT, UPDATE, DELETE **Query optimization** No Yes (Postgres planner pushes conditions to remote server) **Ease of use** Simple for quick queries More setup, but easier for ongoing use **Use case** One-time lookups, quick joins Long-term integration, production workloads

□ In short:

- Use **`dblink`** if you just need a quick lookup across databases.
- Use **`postgres_fdw`** if you need continuous, reliable, and optimized access to remote tables.

Step-by-Step Guide: Implementing `postgres_fdw` in PostgreSQL 15

Now that we understand the difference, let's dive into the **step-by-step process** of setting up and using `postgres_fdw` in PostgreSQL 15.

Step 1: Install the contrib package

If you haven't already installed contrib modules for PostgreSQL 15, do so:

```
sudo yum install postgresql15-contrib



[root@ggnode1 dbs]# sudo yum install postgresql15-contrib    # or: sudo yum install
postgresql15-contrib
ol7_UEKR6
| 3.0 kB  00:00:00
```

```
ol7_addons
| 3.0 kB  00:00:00
ol7_developer
| 3.0 kB  00:00:00
ol7_developer_EPEL
| 3.6 kB  00:00:00
ol7_latest
| 3.6 kB  00:00:00
ol7_optional_latest
| 3.0 kB  00:00:00
ol7_software_collections
| 3.0 kB  00:00:00
pgdg-common/7Server/x86_64/signature
|  665 B  00:00:00
pgdg-common/7Server/x86_64/signature
| 2.9 kB  00:00:00 !!!
pgdg12/7Server/x86_64/signature
|  665 B  00:00:00
pgdg12/7Server/x86_64/signature
| 3.6 kB  00:00:00 !!!
pgdg13/7Server/x86_64/signature
|  665 B  00:00:00
pgdg13/7Server/x86_64/signature
| 3.6 kB  00:00:00 !!!
pgdg14/7Server/x86_64/signature
|  665 B  00:00:00
pgdg14/7Server/x86_64/signature
| 3.6 kB  00:00:00 !!!
pgdg15/7Server/x86_64/signature
|  665 B  00:00:00
pgdg15/7Server/x86_64/signature
| 3.6 kB  00:00:00 !!!
Resolving Dependencies
--> Running transaction check
---> Package postgresql15-contrib.x86_64 0:15.14-1PGDG.rhel7 will be installed
--> Processing Dependency: libpython3.6m.so.1.0()(64bit) for package: postgresql15-
contrib-15.14-1PGDG.rhel7.x86_64
--> Running transaction check
---> Package python3-libs.x86_64 0:3.6.8-21.0.1.el7_9 will be installed
--> Processing Dependency: python(abi) = 3.6 for package: python3-libs-3.6.8-
21.0.1.el7_9.x86_64
--> Running transaction check
---> Package python3.x86_64 0:3.6.8-21.0.1.el7_9 will be installed
--> Processing Dependency: python3-pip for package: python3-3.6.8-
21.0.1.el7_9.x86_64
--> Processing Dependency: python3-setuptools for package: python3-3.6.8-
21.0.1.el7_9.x86_64
--> Running transaction check
---> Package python3-pip.noarch 0:9.0.3-8.0.3.el7 will be installed
---> Package python3-setuptools.noarch 0:39.2.0-10.0.3.el7 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

===================================================================================
===================================================================================
=========
 Package                                         Arch
Version                                       Repository
Size
===================================================================================
===================================================================================
=========
Installing:
 postgresql15-contrib                            x86_64
15.14-1PGDG.rhel7                             pgdg15
722 k
Installing for dependencies:
 python3                                         x86_64
3.6.8-21.0.1.el7_9                            ol7_latest
70 k
 python3-libs                                    x86_64
3.6.8-21.0.1.el7_9                            ol7_latest
7.0 M
```

```
 python3-pip                                      noarch
9.0.3-8.0.3.el7                                 ol7_latest
1.6 M
 python3-setuptools                               noarch
39.2.0-10.0.3.el7                               ol7_latest
629 k

Transaction Summary
=====================================================================================
=====================================================================================
=========
Install  1 Package (+4 Dependent packages)

Total download size: 10 M
Installed size: 50 M
Is this ok [y/d/N]: y
Downloading packages:
(1/5): python3-3.6.8-21.0.1.el7_9.x86_64.rpm
|  70 kB  00:00:00
(2/5): postgresql15-contrib-15.14-1PGDG.rhel7.x86_64.rpm
| 722 kB  00:00:00
(3/5): python3-pip-9.0.3-8.0.3.el7.noarch.rpm
| 1.6 MB  00:00:00
(4/5): python3-setuptools-39.2.0-10.0.3.el7.noarch.rpm
| 629 kB  00:00:00
(5/5): python3-libs-3.6.8-21.0.1.el7_9.x86_64.rpm
| 7.0 MB  00:00:01
-------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------
---------
Total
```

This package contains both `dblink` and `postgres_fdw`.

Step 2: Enable the extension in your database

Log in to PostgreSQL and enable the FDW extension:

```
CREATE EXTENSION postgres_fdw;



postgres=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
postgres=#
```

Verify that it’s installed:

```
\dx postgres_fdw
```

```
postgres=# \dx postgres_fdw
                             List of installed extensions
     Name     | Version | Schema |                    Description
--------------+---------+--------+----------------------------------------------------
--
 postgres_fdw | 1.1     | public | foreign-data wrapper for remote PostgreSQL
servers
(1 row)

postgres=#
```

Step 3: Create a foreign server

Define a connection to the remote PostgreSQL server:

```
CREATE SERVER server_remote FOREIGN DATA WRAPPER postgres_fdw
postgres-# OPTIONS (host '192.168.50.11', dbname 'postgres', port '5432');



postgres=# CREATE SERVER server_remote FOREIGN DATA WRAPPER postgres_fdw
postgres-# OPTIONS (host '192.168.50.11', dbname 'postgres', port '5432');
CREATE SERVER
postgres=#
```

Here:

- **host** = IP address of the remote server
- **dbname** = remote database name
- **port** = default PostgreSQL port (5432)

Step 4: Create a user mapping

Map the current user to the remote user by providing login credentials:

```
CREATE USER MAPPING FOR current_user SERVER server_remote
OPTIONS (user 'postgres', password 'oracle123');
```

```
postgres=# CREATE USER MAPPING FOR current_user SERVER server_remote
postgres-# OPTIONS (user 'postgres', password 'oracle123');
CREATE USER MAPPING
postgres=#
```

This ensures that when the local user connects to the foreign server, it uses the specified username and password.

Step 5: Import schema or create foreign tables

At this point, you can either **import an entire schema** from the remote database or manually define a foreign table.

Option A: Import all tables from a schema

```
IMPORT FOREIGN SCHEMA public FROM SERVER server_remote
INTO public;



postgres=# IMPORT FOREIGN SCHEMA public FROM SERVER server_remote
postgres-# INTO public;
IMPORT FOREIGN SCHEMA
postgres=#
```

This automatically creates local foreign tables for everything in the remote `public` schema.

Viewing Local and Foreign Tables

After setting up `postgres_fdw` and importing or creating foreign tables, you can list all relations (tables, views, sequences, etc.) in your current schema. In PostgreSQL, the command for this is:

```
 \d+
```

Example output:

```
postgres=# \d+
                                                  List of relations
 Schema |           Name            |     Type      |  Owner   | Persistence | Access
method |    Size     | Description
--------+---------------------------+---------------+----------+-------------+--------
-------+------------+-------------
 public | items                     | foreign table | postgres | permanent   |
| 0 bytes    |
 public | orders                    | foreign table | postgres | permanent   |
| 0 bytes    |
 public | products                  | table         | postgres | permanent   | heap
| 977 MB     |
 public | products_product_id_seq | sequence      | postgres | permanent   |
| 8192 bytes |
(4 rows)

postgres=#
```

What This Means

- **`items` (foreign table):**
  This is a table that resides on the remote PostgreSQL server but is accessible locally through `postgres_fdw`.

- **`orders` (foreign table):**
  Another remote table, seamlessly integrated into the local database. Queries on this table are executed against the remote database.

- **`products` (local table):**
  A regular table that exists locally in the `public` schema.

- **`products_product_id_seq` (sequence):**
  A sequence object used to generate IDs for the `products` table.

Key Takeaway

With `postgres_fdw`, your local PostgreSQL instance can contain a **mix of local and remote objects**.

- To your application, these look almost the same.

- The big difference is that **foreign tables delegate operations to the remote database**.

This setup makes it possible to query and join data across multiple databases without complex ETL processes.

Option B: Define a specific foreign table

If you want to add new tables after importing a foreign schema, you cannot re-import the foreign tables. Instead, you can use this option to add additional tables.

```
postgres=# IMPORT FOREIGN SCHEMA public FROM SERVER server_remote
postgres-# INTO public;
ERROR:  relation "items" already exists
CONTEXT:  importing foreign table "items"
postgres=#



CREATE FOREIGN TABLE items_remote (
    item_id   INT,
    quantity  INT,
    price     NUMERIC(10,2)
)
SERVER server_remote
OPTIONS (table_name 'items');



postgres=# CREATE FOREIGN TABLE items_remote (
postgres(#     item_id   INT,
postgres(#     quantity  INT,
postgres(#     price     NUMERIC(10,2)
postgres(# )
postgres-# SERVER server_remote
postgres-# OPTIONS (table_name 'items');
CREATE FOREIGN TABLE
postgres=#
```

Here, `items_remote` is the local representation of the remote `items`table or any new tables.

```
postgres=# \d+
                                                  List of relations
 Schema |          Name            |     Type      |  Owner   | Persistence | Access
method |    Size    | Description
--------+--------------------------+---------------+----------+-------------+--------
-------+------------+-------------
 public | items                    | foreign table | postgres | permanent   |
| 0 bytes    |
 public | items_remote             | foreign table | postgres | permanent   |
| 0 bytes    |
 public | orders                 | foreign table | postgres | permanent   |
| 0 bytes     |
 public | products               | table         | postgres | permanent   | heap
| 977 MB      |
 public | products_product_id_seq | sequence      | postgres | permanent   |
| 8192 bytes |
(5 rows)

postgres=#
```

Step 6: Query the foreign table

Now you can query the remote table as if it were local:

```
SELECT * FROM items WHERE price > 200;



postgres=# SELECT * FROM items WHERE price > 200;

 item_id  | quantity | price
----------+----------+--------
  9093121 |      297 | 395.80
  9093122 |      640 | 226.10
  9093125 |       18 | 222.13
  9093128 |      739 | 493.85
  9093131 |      613 | 339.56
  9093133 |      917 | 461.97
  9093135 |      719 | 433.66
  9093137 |      211 | 212.29
  9093138 |      732 | 212.20
  9093140 |      319 | 295.23
```

Step 7: Perform DML operations

Unlike `dblink`, `postgres_fdw` allows you to modify data on the remote server.

- **Delete data:**

```
DELETE FROM items WHERE item_id = 9093129;



postgres=# DELETE FROM items WHERE item_id = 9093129;
DELETE 1
postgres=#
```

- **Insert data:**

```
INSERT INTO items VALUES (9093129, 500, 250.00);



postgres=# INSERT INTO items VALUES (9093129, 500, 250.00);
INSERT 0 1
postgres=#
```

- **Update data:**

```
UPDATE items SET price = 199.99 WHERE item_id = 9093121;



postgres=# UPDATE items SET price = 199.99 WHERE item_id = 9093121;
UPDATE 1
postgres=#
```

All changes are applied directly to the remote database.

□ Practical Considerations

1. **Performance:**

- `postgres_fdw` pushes down WHERE conditions to the remote server, reducing network traffic.
- Joins between local and foreign tables may be less efficient, so use filters.

2. **Security:**

- Avoid storing plain-text passwords in user mappings. Use role-based authentication or password files.

3. **Use cases:**

- Ideal for distributed reporting, microservices accessing shared data, or consolidating analytics.
- Not a replacement for replication — use logical or streaming replication for high availability.

□ Final Thoughts

- `dblink` is the older, simpler way: best for **occasional queries**.
- `postgres_fdw` is the modern, integrated way: best for **production systems where remote tables should behave like local ones**.

With the steps above, you can seamlessly connect PostgreSQL databases using `postgres_fdw` and perform both read and write operations across multiple systems.