Semestral Project NI-PDB 2024/2025

Foreign Data Wrappers in PostgreSQL

Shchapaniak Andrei

FIT CVUT

December 3, 2024

1 Introduction

In today's data-driven world, organizations face a challenge with integrating vast amounts of data from diverse sources. As businesses grow and evolve, they often accumulate data across multiple platforms, databases, and even geographical locations. The fragmentation of data into multiple "silos" can significantly hinder an organization's ability to access and utilize its information efficiently and in real-time. Traditional data integration methods often involve complex processes or the creation of data warehouses, which can be time-consuming, resource-intensive, and may not provide real-time access.

The need for standardized external data access led to the development of the SQL/Management of External Data (`SQL/MED`) extension to the SQL standard in 2003. This extension introduced the concept of Foreign Data Wrappers (`FDWs`), offering a unified interface for accessing external data sources. PostgreSQL introduced FDWs in version 9.1 (2011). It provided a more efficient and flexible approach that can help to overcome certain challenges.

FDWs address the challenges of data silos by enabling seamless access to distributed data, eliminating the need for complex migrations or duplications. In the sections that follow, we'll delve into their core functionalities and different trade-offs of their usage. Also, the last section describes how to configure and use them with various types of databases.

2 Understanding FDWs

Foreign Data Wrappers represent a sophisticated abstraction layer that enables a database system to interact with external data sources as if they were native tables. At their core, FDWs implement a set of predefined interfaces that handle the complexity of external data access, query planning, and execution. This architecture consists of three primary components: `FDW Handler`, which implements the interface functions; `Foreign Server`, which maintains connection information; and `Foreign Tables`, which map external data to local table definitions.

2.1 Virtual Table Mechanism

The fundamental principle behind FDWs is virtualizing external data access through a sophisticated proxy mechanism. Rather than physically storing external data within the database, FDWs create virtual tables (interfaces) that serve as proxies to the actual data sources. These virtual tables are essentially metadata definitions that describe the structure of the remote data and contain the necessary connection information. When a query is executed against a virtual table, the FDW establishes a network connection to the remote data source

using TCP/IP protocol[1] and credentials stored in the foreign server definition.

The beauty of this virtual table approach lies in its transparency to end users. Applications and queries interact with foreign tables using standard SQL syntax, completely unaware that they're accessing remote data. This abstraction simplifies application development and provides the flexibility to modify underlying data sources without impacting existing applications.

2.2 Multi-Layer Processing Architecture

The proxy mechanism operates through a multi-layered approach:

- `Query Interface Layer` - when a query references a foreign table, the PostgreSQL executor first interacts with the FDW's handler functions. These functions translate the PostgreSQL query operations into a format suitable for the remote system.
- `Connection Management Layer` - the FDW maintains a connection pool to the remote data source, reusing existing connections when possible to reduce overhead. This layer handles authentication, connection lifecycle, and session management.
- `Data Transfer Layer` - as data is requested, it flows through network sockets between the remote source and PostgreSQL. The FDW implements buffering and streaming mechanisms to optimize data transfer, particularly for large result sets. Rather than transferring entire tables, the FDW retrieves only the rows and columns necessary to satisfy the current query.
- `Cache Management Layer` - some FDW implementations maintain a metadata cache containing information about remote table structures, statistics, and frequently accessed data. This reduces network roundtrips for repeated queries while ensuring cache invalidation when remote data changes.

2.3 Data Type Management and Translation

A critical challenge in cross-database communication is managing different data type systems. FDWs implement comprehensive type mapping mechanisms to handle conversions between databases automatically. This extends beyond basic data types to include complex structures like arrays and custom types. For example, when PostgreSQL communicates with MySQL, the FDW automatically translates between corresponding types (such as MySQL's `DATETIME` to PostgreSQL's `TIMESTAMP`), handling differences in character encodings and numeric precision while maintaining data integrity. This automatic translation happens transparently, allowing users to work with data without concerning themselves with underlying type differences.

2.4 Key Advantages

Using FDWs has several key benefits for managing data:

- `Zero-Copy Access` - data remains in its original location and is only transferred when explicitly queried.

[1] https://www.techtarget.com/searchnetworking/definition/TCP-IP

- `Real-Time Consistency` - changes in the source data are immediately reflected in queries against the virtual tables.
- `Resource Efficiency` - storage requirements are minimal, limited to metadata and temporary query results.
- `Transparent Access` - applications can interact with foreign tables using standard SQL, unaware of the underlying proxy mechanism.

3 Technical Implementation and Operation

The technical implementation of FDWs involves several intricate components working together to enable seamless cross-database operations. This section examines the core mechanisms behind its functionality, including the query processing pipeline, performance optimization strategies, and practical limitations.

3.1 Query Processing Pipeline

The query processing mechanism showcases architectural sophistication through a multi-phase execution process. When a query involving foreign tables is received, the PostgreSQL planner orchestrates the execution strategy through distinct phases. During query analysis, it identifies which operations should be executed locally versus remotely by evaluating table relationships, join conditions, and query predicates. The planner then generates an optimized execution plan, making critical decisions about join ordering and predicate pushdown opportunities to minimize data movement. Finally, in the remote query construction phase, the FDW translates the relevant PostgreSQL query segments into the remote system's SQL dialect, handling differences in function availability and optimization capabilities. This systematic approach ensures efficient query execution while abstracting the complexity of cross-database operations from the user.

3.2 Performance Optimization

PostgreSQL FDW implementations require thinking like an application developer rather than relying solely on PostgreSQL's automatic optimization. The key strategies for maximizing efficiency include:

- `Query Containerization`: Common Table Expressions (`CTE`s) serve as a powerful tool for containing remote queries. They enable more efficient querying by sending targeted, filtered requests to the remote server instead of retrieving all records. This approach is particularly valuable when data needs to be pre-filtered before joining operations.
- `Subquery Optimization`: The implementation of subqueries can be significantly improved by utilizing the **ANY** operator in combination with **ARRAY_AGG**, rather than relying on **IN** clauses. This method ensures that query evaluation occurs before data is transmitted to the remote server, leading to more efficient query execution.
- `Fetch Size Configuration`: The default fetch size of 100 rows often proves inadequate for most use cases. By increasing the fetch size through **ALTER SERVER** commands, the system can reduce the number of required fetch operations, resulting in improved performance for large data retrievals.

- `Local Caching Strategies`: Two primary approaches exist for local caching. Materialized Views offer the simplest solution, enabling local data caching with indexing capabilities, though requiring manual refresh for updates. For more advanced needs, Cache Tables with **MERGE** (available in PostgreSQL 15+) provide incremental updates of large datasets and greater control over the update process, effectively serving as a custom ETL solution.

- `Join Type Classification`: Understanding and optimizing three distinct join types is crucial. Local joins execute entirely on the querying server, providing optimal performance. Remote joins process on the foreign server between tables on the same foreign server, maintaining reasonable efficiency. Cross-server joins, involving multiple servers, represent the least efficient option and should be minimized where possible.

Implementation of these strategies can significantly enhance FDW query performance while maintaining data consistency and reducing unnecessary data transfer between servers[2].

3.3 Limitations and Practical Challenges

While FDWs provide powerful data integration capabilities, they face several critical limitations that impact their practical implementation. Network latency represents the primary challenge, particularly in complex queries joining local and foreign tables where multiple server round-trips are required. The PostgreSQL query planner's effectiveness is compromised by limited access to foreign table statistics, often resulting in suboptimal execution plans, especially for large-scale operations and complex joins. Additionally, maintaining transactional integrity across heterogeneous databases proves challenging, as most FDW implementations cannot fully support distributed transactions with two-phase commit protocols. Data type compatibility further complicates integration, particularly with specialized database types that lack direct equivalents across systems, potentially affecting data integrity if not properly managed.

4 Major PostgreSQL FDWs

PostgreSQL's FDW ecosystem includes several powerful implementations, each designed to address specific data integration challenges. While many wrappers exist, three implementations stand out for their practical utility:

4.1 File FDW

The file foreign data wrapper represents one of PostgreSQL's most straightforward yet powerful data integration tools. Built into PostgreSQL's core distribution, it transforms simple files into queryable database tables, enabling seamless integration of file-based data sources into database operations. The wrapper offers an elegant combination of simplicity and versatility. While one of its popular uses is handling CSV files, the wrapper also supports various text file formats through configurable delimiters, quote characters, and encoding specifications.

However, this simplicity comes with certain constraints. Wrapper operates primarily in read-only mode, meaning direct modifications to the underlying files through SQL operations

[2] https://www.crunchydata.com/blog/performance-tips-for-postgres-fdw

aren't supported. Performance depends heavily on the underlying file system's characteristics, and dealing with malformed data requires careful error-handling strategies.

4.2 MongoDB FDW

MongoDB's foreign data wrapper bridges the gap between PostgreSQL's rigid relational structure and MongoDB's flexible document-oriented model. It tackles the complex challenge of mapping MongoDB's rich document structures to PostgreSQL's tabular format while preserving the ability to leverage MongoDB's unique features.

The wrapper's sophisticated mapping engine translates between these disparate data models transparently. It handles nested document structures, arrays, and MongoDB's dynamic schemas, presenting them in a format that PostgreSQL applications can easily consume. This translation layer enables organizations to leverage MongoDB's flexibility for certain data while maintaining PostgreSQL as their primary analytical engine.

4.3 MySQL FDW

The MySQL foreign data wrapper serves a crucial role in environments where PostgreSQL needs to coexist with MySQL databases. This integration proves particularly valuable during database migrations, in heterogeneous database environments, or when different applications require different database engines.

Wrapper handles the complexities of cross-database communication transparently. It manages the intricacies of data type conversion between MySQL and PostgreSQL, dealing with differences in SQL dialects, and maintaining efficient connection pooling. The wrapper's query planner works to optimize operations by pushing down appropriate predicates and joins to MySQL, reducing data transfer overhead.

5 Alternatives to FDWs

When considering data integration strategies, organizations must carefully evaluate various approaches beyond FDWs. Each integration method offers distinct advantages and trade-offs, making the selection process crucial for long-term success. Understanding these alternatives helps organizations make informed decisions based on their specific requirements, such as real-time access needs, performance demands, and implementation complexity. The following analysis examines the most prevalent integration approaches, highlighting their strengths and optimal use cases to provide a comprehensive comparison with FDW solutions.

5.1 ETL Processes

Extract, Load, Transform (`ETL`) represents a traditional and well-established approach to data integration. This process extracts data from various sources, applies necessary transformations, and loads it into a target system, typically a data warehouse. Modern ETL tools support both batch and real-time processing patterns, offering significant flexibility in data handling.

The strength of ETL lies in its ability to handle complex data transformations and maintain historical records. Organizations can implement sophisticated data cleaning, enrichment, and aggregation processes during the transformation phase. The resulting data

warehouse provides optimized structures for analytical queries, often delivering superior query performance compared to real-time integration approaches.

However, ETL processes introduce data latency by their very nature. The time gap between data creation in source systems and availability in the target system can range from minutes to hours, depending on the implementation.

5.2 Data Virtualization

Data virtualization technology creates an abstraction layer that provides unified access to diverse data sources without physical data movement. This approach shares some conceptual similarities with FDWs but typically offers broader functionality and more hig data integration capabilities.

Modern data virtualization platforms provide advanced features for data transformation, caching, and query optimization. They can integrate data from various sources including databases, flat files, APIs, and cloud services. The abstraction layer handles differences in data formats, protocols, and access patterns, presenting a unified interface to applications.

The primary limitation of data virtualization emerges in query performance, particularly for complex operations across multiple data sources. While caching strategies can mitigate some performance issues, the fundamental challenges of distributed query execution remain. Network latency and bandwidth constraints can impact query response times, especially when dealing with large data volumes.

5.3 API-based Integration

API-based integration is a modern way to handle data access that works especially well with cloud and microservices systems. It works by creating or using APIs that let applications directly access data sources in real-time. What makes this approach powerful is that it provides standardized ways to connect to data and can handle both immediate and delayed data requests.

However, there are some important challenges to consider. Developers often need to write custom code for each data source they want to connect to, and it can be tricky to make complex data queries work efficiently when dealing with multiple APIs. Another issue is that older systems might not have built-in API support, which means extra work is needed to make them compatible.

5.4 Comparative Analysis

Aspect	FDWs	ETL	Data Virtualization	API Integration
Real-time Access	Yes	No	Yes	Yes
Query Performance	Moderate	High	Moderate	Varies
Data Duplication	Low	High	Low	Low
Setup Complexity	Moderate	High	Moderate	Varies
Flexibility	High	Low	High	High
Scalability	High	High	Moderate	High
Maintenance Effort	Moderate	High	Moderate	High
Development Cost	Moderate	High	Moderate	High
Data Consistency	Real-time	Periodic	Real-time	Real-time

Table 1: Comparison of Data Integration Approaches

6 Practical examples

This section describes the detailed implementation and configuration of PostgreSQL FDWs in a distributed database environment. The system integrates multiple data sources, demonstrating practical applications of FDW technology.

6.1 System Architecture

The implementation creates a distributed database system that combines:

- `PostgreSQL`: Primary database for order management.
- `MySQL`: Product catalog management.
- `MongoDB`: Customer information storage.
- `CSV Files`: Tax rate configuration and rules.

Infrastructure Components

The system utilizes Docker containers for service isolation and easy deployment:

```
services:
  postgres:
    image: postgres:15

  mongodb:
    image: mongo:7.0

  mysql:
    image: mysql:8.0
```

6.2 FDWs Installation and Setup

The installation begins with a custom PostgreSQL image that includes all necessary extensions:

MySQL

The extension `postgresql-15-mysql-fdw` is installed via the PostgreSQL package repositories, providing native MySQL connectivity and automatic data type mapping. There is an alternative approach that involves manual installation and configuration. For more details about manual installation, you can refer to the official git repository[3].

MongoDB

The FDW extension for MongoDB requires manual compilation:

```
git clone https://github.com/EnterpriseDB/mongo_fdw.git
cd mongo_fdw
chmod +x autogen.sh
./autogen.sh

export MONGO_FDW_SOURCE_DIR=/docker-entrypoint-initdb.d/mongo_fdw
export PKG_CONFIG_PATH=$MONGO_FDW_SOURCE_DIR/mongo-c-driver/src/libmongoc/src
make USE_PGXS=1
make USE_PGXS=1 install
```

You can read more about it directly from the git repository of this extension for PostgreSQL[4].

CSV File

This wrapper is included in the PostgreSQL core and requires only SQL activation:

```
CREATE EXTENSION file_fdw;
```

6.3 Database Configuration

Extension Setup

Initialize required extensions:

```
CREATE EXTENSION IF NOT EXISTS mysql_fdw;
CREATE EXTENSION IF NOT EXISTS mongo_fdw;
CREATE EXTENSION IF NOT EXISTS file_fdw;
```

[3] https://github.com/EnterpriseDB/mysql_fdw
[4] https://github.com/EnterpriseDB/mongo_fdw

Server Definitions and User Mappings

FDWs require server definitions to establish connections with external data sources. These server objects act as connection endpoints, specifying how PostgreSQL should communicate with foreign data sources. Each server definition includes essential connection parameters and authentication settings. Also, we must establish user mappings that define the credentials for connection to the foreign data source.

MySQL Server

```
CREATE SERVER mysql_server
    FOREIGN DATA WRAPPER mysql_fdw
    OPTIONS (host 'pdb_mysql', port '3306');

CREATE USER MAPPING FOR postgres
    SERVER mysql_server
    OPTIONS (username 'mysql', password 'mysql');
```

MongoDB Server

```
CREATE SERVER mongo_server
    FOREIGN DATA WRAPPER mongo_fdw
    OPTIONS (address 'pdb_mongo', port '27017', authentication_database 'admin'
        );

CREATE USER MAPPING FOR postgres
    SERVER mongo_server
    OPTIONS (username 'root', password 'mongodb');
```

File CSV Server

```
CREATE SERVER csv_server FOREIGN DATA WRAPPER file_fdw;
```

This approach helps to keep authentication details separate from server definitions and allows different PostgreSQL users to have different permissions on foreign servers.

Foreign Tables and User Mappings

Foreign tables in PostgreSQL act as proxies that allow access to data stored in external data sources as if they were regular tables. However, unlike regular tables, foreign tables don't store any data themselves - they serve as interfaces to read and manipulate data in their respective external sources.

Products Table (MySQL)

```
CREATE FOREIGN TABLE products (
    product_id INTEGER,
    name VARCHAR(100),
    price DECIMAL(10,2),
    category VARCHAR(50),
    stock INTEGER
) SERVER mysql_server
```

```
OPTIONS (dbname 'product_db', table_name 'products');
```

Customers Table (MongoDB)

```
CREATE FOREIGN TABLE customers (
    name VARCHAR(100),
    email VARCHAR(100),
    addresses TEXT[],
    tax_region VARCHAR(50)
) SERVER mongo_server
OPTIONS (database 'demo', collection 'customers');
```

Tax Rates Table (CSV)

```
CREATE FOREIGN TABLE tax_rates (
    region VARCHAR(50),
    category VARCHAR(50),
    rate DECIMAL(5,2),
    rules TEXT
) SERVER csv_server
OPTIONS (filename '/data/tax_rates.csv', format 'csv', header 'true');
```

6.4 Practical Usage Examples

1. Query

```
EXPLAIN ANALYZE
SELECT
    c.name,
    p.name as product_name,
    p.price,
    tr.rate as tax_rate,
    (p.price * tr.rate / 100) as tax_amount
FROM customers c
JOIN tax_rates tr ON c.tax_region = tr.region
CROSS JOIN products p
WHERE p.category = tr.category;
```

1. Output

```
Merge Join (cost=1063.66..2066.66 rows=25 width=496) (actual time=3.804..3.893 rows=6
    loops=1)
  Merge Cond: ((p.category)::text = (tr.category)::text)
  -> Foreign Scan on products p (cost=25.00..1025.00 rows=1000 width=352) (actual
     time=0.753..0.784 rows=3 loops=1)
       Remote server startup cost: 25
  -> Sort (cost=1038.66..1038.67 rows=5 width=348) (actual time=3.033..3.054 rows=4
     loops=1)
       Sort Key: tr.category
       Sort Method: quicksort Memory: 25kB
       -> Nested Loop (cost=25.00..1038.60 rows=5 width=348) (actual time
          =0.715..3.012 rows=2 loops=1)
            Join Filter: ((c.tax_region)::text = (tr.region)::text)
            Rows Removed by Join Filter: 8
            -> Foreign Scan on tax_rates tr (cost=0.00..1.10 rows=1 width=248) (
               actual time=0.031..0.082 rows=5 loops=1)
                 Foreign File: /data/tax_rates.csv
                 Foreign File Size: 233 b
            -> Foreign Scan on customers c (cost=25.00..1025.00 rows=1000 width=336)
                (actual time=0.561..0.569 rows=2 loops=5)
                 Foreign Namespace: demo.customers
Planning Time: 0.810 ms
Execution Time: 6.057 ms
```

We can see that PostgreSQL's optimizer chooses a merge join strategy to combine the distributed data efficiently. Each foreign scan operation incurred a startup cost of 25 units, representing the overhead of establishing connections to remote servers. The planner's row estimates were higher than actual values, which is typical when working with foreign tables where statistics are less precise. For instance, the scan of the products table was estimated for 1000 rows but retrieved only 3 rows.

2. Query

```
EXPLAIN ANALYZE
SELECT * FROM products p
JOIN customers c ON p.category = 'Electronics'
WHERE c.tax_region = 'US-CA';
```

2. Output

```
Nested Loop (cost=50.00..14552.50 rows=1000000 width=946) (actual time=1.157..1.212
   rows=3 loops=1)
  -> Foreign Scan on products p (cost=25.00..1025.00 rows=1000 width=360) (actual
     time=0.561..0.576 rows=3 loops=1)
       Remote server startup cost: 25
  -> Materialize (cost=25.00..1030.00 rows=1000 width=586) (actual time=0.196..0.201
     rows=1 loops=3)
       -> Foreign Scan on customers c (cost=25.00..1025.00 rows=1000 width=586) (
          actual time=0.580..0.584 rows=1 loops=1)
            Foreign Namespace: demo.customers
Planning Time: 0.570 ms
Execution Time: 2.631 ms
```

Each foreign scan operation took a startup cost of 25 units. The presence of 2 foreign scans demonstrates that our query is working across different databases, using FDW to seamlessly join data from separate sources. The materialization of the customer's data (shown by the Materialize operation) indicates that PostgreSQL optimized the remote data access by caching the foreign table's results for repeated use in the nested loop join.

7 Conclusion

This paper has explored Foreign Data Wrappers as a powerful mechanism for database integration, examining both their theoretical foundations and practical implementations. The examination of its architecture revealed its effective approach to data integration, providing transparent access to external data sources while managing connection pooling, query optimization, and type mapping. The practical implementation demonstrated the capabilities of FDWs in distributed queries. While these wrappers may not offer the performance of dedicated ETL processes or the flexibility of API integration, they provide a practical compromise for scenarios where real-time data access and SQL-based querying are primary requirements.

However, organizations should carefully consider performance requirements and transaction management needs when implementing FDW-based solutions. It could be beneficial especially, when deal with[5]:

- Administrative tasks and reporting where real-time performance isn't critical.
- Incremental data synchronization and migration are needed, particularly when full ETL processes would be overengineered.
- Small-scale data operations and infrequent queries where the overhead of maintaining multiple connections would be more costly than the performance impact.

Future research should concentrate on detailed performance evaluation across diverse use cases, examining how FDWs perform with varying data volumes, query complexities, and data distribution patterns. This analysis would include measuring execution times, resource consumption, and scalability characteristics under different workloads and concurrency levels. Understanding these performance patterns would help establish optimal configuration guidelines and identify scenarios where FDWs offer the most value compared to alternative integration approaches.

[5] https://dev.to/k1hara/exploring-postgresql-foreign-data-wrappers-fdw-3l37