The Internals of PostgreSQL

for database administrators and system developers

Chapter 4

Foreign Data Wrappers and Parallel Query

This chapter will describe two technically interesting and practical features: Foreign Data Wrappers(FDW) and Parallel Query.

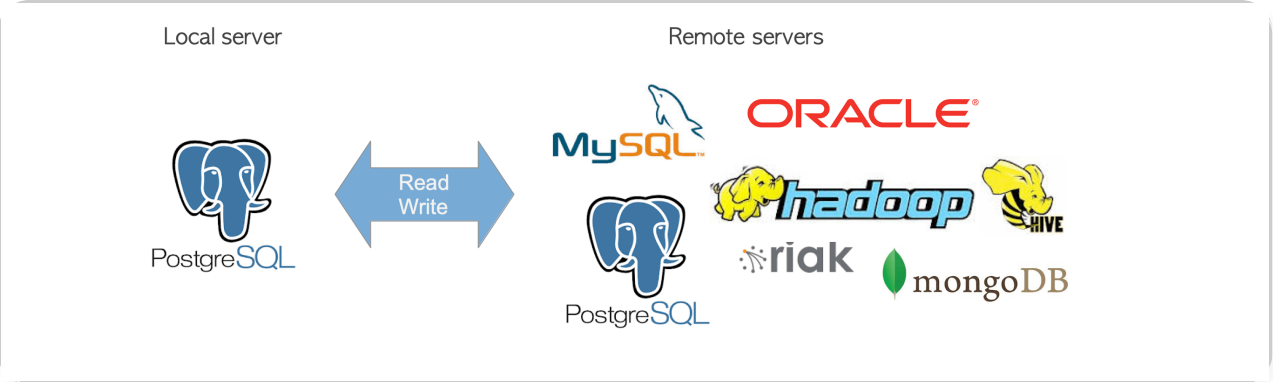
Currently, Section 4.1. FDW is only provided; Section 4.2. Parallel Query is under construction.

4.1. Foreign Data Wrappers (FDW)

In 2003, a specification to access remote data, called SQL Management of External Data (SQL/MED), was added to the SQL standard. This feature has been developing by PostgreSQL to realize a portion of SQL/MED since version 9.1.

In SQL/MED, a table on a remote server is called a *foreign table*. PostgreSQL's **Foreign Data Wrappers (FDW)** are that use SQL/MED to manage foreign tables which are similar to local tables.

Fig. 4.1. Basic concept of FDW.



After installing the necessary extension and making the appropriate settings, you can access the foreign tables on the remote servers. For example, suppose there are two remote servers, namaly, postgresql and mysql, which have *foreign_pg_tbl* table and *foreign_my_tbl table*,

respectively. In this exmaple, you can access the foreign tables from the local server by issuing the SELECT queries as shown below.

```
localdb=# -- foreign_pg_tbl is on the remote
 postgresql server.
localdb-# SELECT count(*) FROM foreign_pg_tbl
;
 count
-------
 20000

localdb=# -- foreign_my_tbl is on the remote
 mysql server.
localdb-# SELECT count(*) FROM foreign_my_tbl
;
 count
-------
 10000
```

Moreover, you can execute the join operation with the foreign tables stored in different servers which are similar to the local tables.

```
localdb=# SELECT count(*) FROM foreign_pg_tbl
AS p, foreign_my_tbl AS m WHERE p.id = m.id;
 count
-------
 10000
```

Many FDW extensions have been developed and listed in Postgres wiki. However, almost all

extensions are not properly maintained except for postgres_fdw, which is officially developed and maintained by the PostgreSQL Global Development Group as an extension to access a remote PostgreSQL server.

PostgreSQL's FDW is described in detail in the following sections. Section 4.1.1 presents an overview of the FDW in PostgreSQL. Section 4.1.2 describes how the postgres_fdw extension performs.

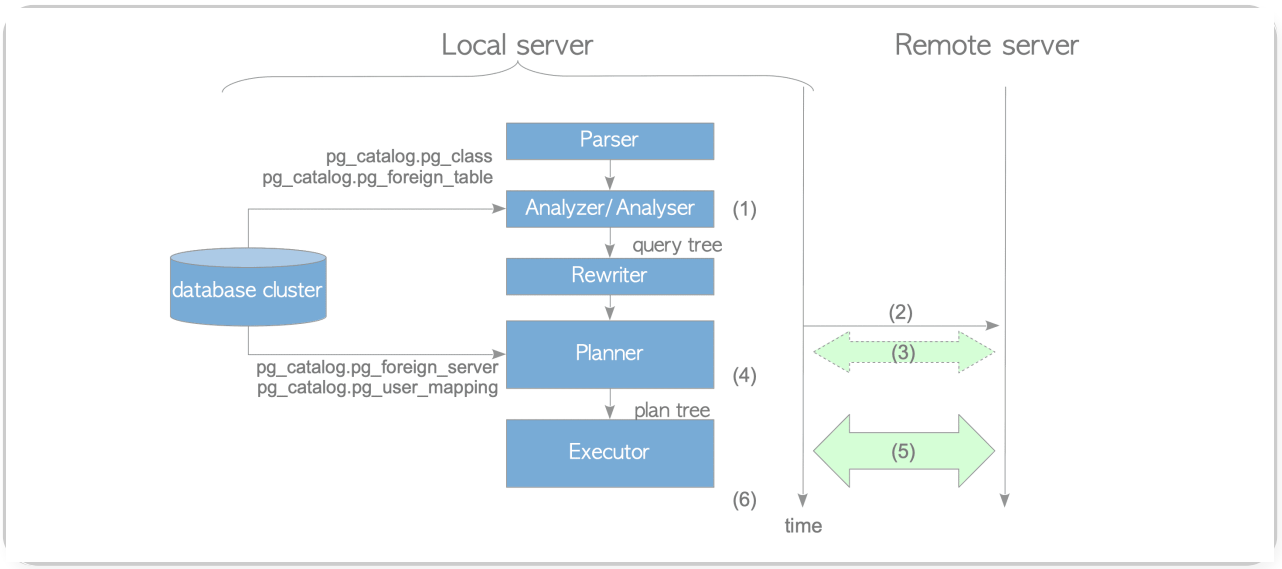
4.1.1. Overview

To use the FDW feature, you need to install the appropriate extension and execute setup commands, such as CREATE FOREIGN TABLE, CREATE SERVER and CREATE USER MAPPING (for details, refer to the official document).

After providing the appropriate setting, the functions defined in the extension are invoked during query processing to access the foreign tables.

Fig.4.2 briefly describes how FDW performs in PostgreSQL.

Fig. 4.2. How FDWs perform.



(1) The analyzer/analyser creates the query tree of the input SQL.

(2) The planner (or executor) connects to the remote server.

(3) If the use_remote_estimate option is *on* (the default is *off*), the planner executes EXPLAIN commands for estimating the cost of each plan path.

(4) The planner creates the plain text SQL statement from the plan tree which is internally called *deparesing*.

(5) The executor sends the plain text SQL statement to the remote server and receives the result.

The executor then processes the received data if necessary. For example, if the multi-table query is

executed, the executor perfoms the join processing of the received data and other tables.

The details of each processing are described in the following sections.

4.1.1.1. Creating a Query Tree

The analyzer/analyser creates the query tree of the input SQL using the definitions of the foreign tables, which are stored in the pg_catalog.pg_class and pg_catalog.pg_foreign_table catalogs using the CREATE FOREIGN TABLE or IMPORT FOREIGN SCHEMA command.

4.1.1.2. Connecting to the Remote Server

To connect to the remote server, the planner (or executor) uses the specific library to connect to the remote database server. For example, to connect to the remote PostgreSQL server, postgres_fdw uses the libpq. To connect to the mysql server, mysql_fdw, which is developed by EnterpriseDB, uses the libmysqlclient.

The connection parameters, such as username, server's IP address and port number, are stored in the pg_catalog.pg_user_mapping and

pg_catalog.pg_foreign_server catalogs using the CREATE USER MAPPING and CREATE SERVER commands.

4.1.1.3. Creating a Plan Tree Using EXPLAIN Commands (Optional)

PostgreSQL's FDW supports the feature to obtain statistics of the foreign tables to estimate the plan tree of a query, which are used by some FDW extensions, such as postgres_fdw, mysql_fdw, tds_fdw and jdbc2_fdw.

If the *use_remote_estimate* option is set to *on* using the ALTER SERVER command, the planner queries the cost of plans to the remote server by executing the EXPLAIN command; otherwise, the embedded constant values are used by default.

```
localdb=# ALTER SERVER remote_server_name OPT
IONS (use_remote_estimate 'on');
```

Although, some extensions use the values of the EXPLAIN command, only postgres_fdw can reflect the results of the EXPLAIN commands because the PostgreSQL's EXPLAIN command returns both the start-up and total costs.

The results of the EXPLAIN command could not be used by other DBMS fdw extensions for planning. For example, mysql's EXPLAIN command only returns the estimated number of rows; however, PostgreSQL's planner needs more information to estimate the cost as described in Chapter 3.

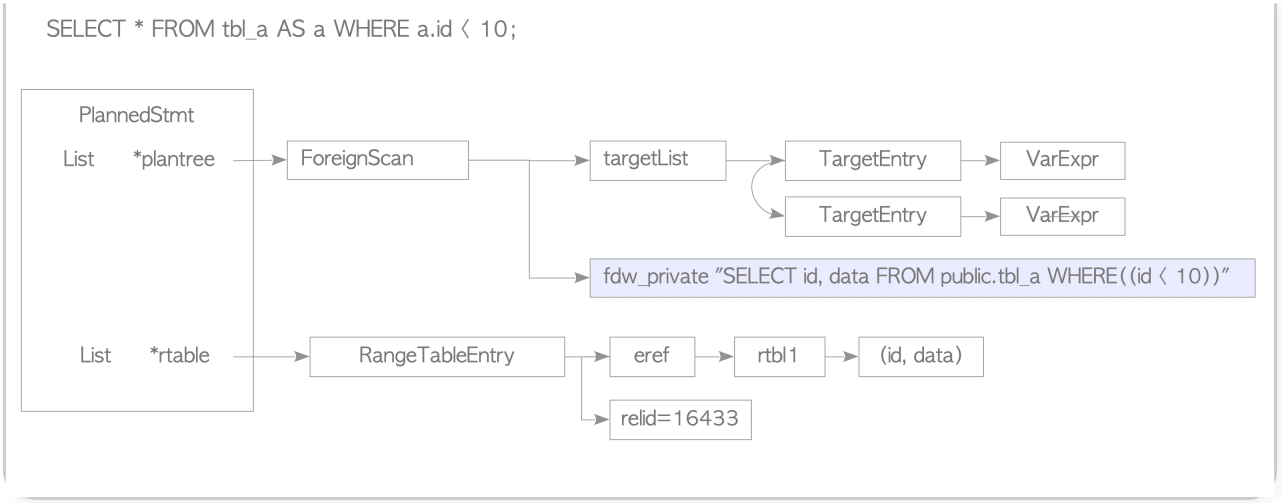
4.1.1.4. Deparesing

To generate the plan tree, the planner creates the plain text SQL statement from the plan tree's scan paths of the foreign tables. For example, Fig. 4.3 shows the plan tree of the following SELECT statement.

```
localdb=# SELECT * FROM tbl_a AS a WHERE a.id
< 10;
```

Fig.4.3 shows that the ForeignScan node, which is linked from the plan tree of the PlannedStmt, stores a plain SELECT text. Here, postgres_fdw recreates a plain SELECT text from the query tree that has been created by parsing and analysing, which is called **deparsing** in PostgreSQL.

Fig. 4.3. Example of the plan tree that scans a foreign table.



The use of mysql_fdw recreates a SELECT text for MySQL from the query tree. The use of redis_fdw or rw_redis_fdw creates a SELECT command.

4.1.1.5. Sending SQL Statements and Receiving Result

After deparsing, the executor sends the deparsed SQL statements to the remote server and receives the result.

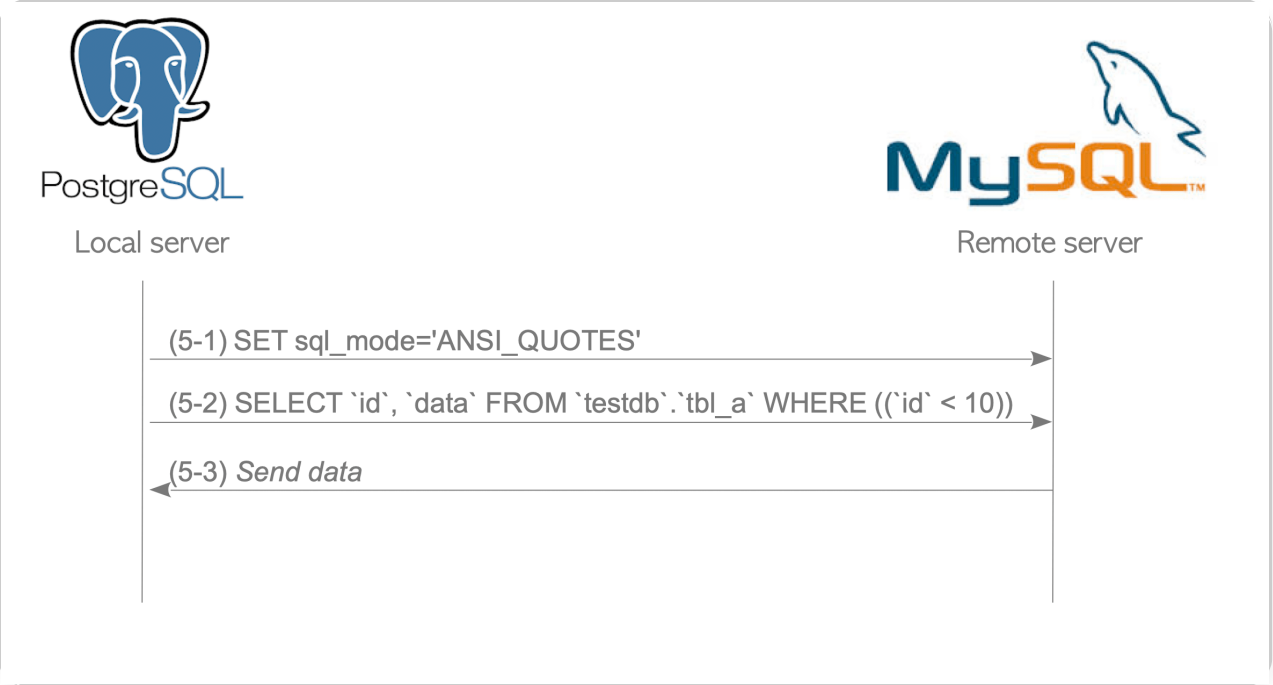
The method for sending the SQL statements to the remote server depends on the developer of each extension. For example, mysql_fdw sends the SQL statements without using a transaction. The typical sequence of SQL statements to execute a SELECT query in mysql_fdw is shown below (Fig. 4.4).

(5-1) Set the SQL_MODE to 'ANSI_QUOTES'.

(5-2) Send a SELECT statement to the remote server.

(5-3) Receive the result from the remote server.
Here, mysql_fdw converts the result to readable data by PostgreSQL.
All FDW extensions implemented the feature that converts the result to PostgreSQL readable data.

Fig. 4.4. Typical sequence of SQL statements to execute a SELECT query in mysql_fdw



The actual log of the remote server can be found here; the statements received by the remote server are shown.

In postgres_fdw, the sequence of SQL commands is complicated. The typical sequence of SQL

statements to execute a SELECT query in postgres_fdw is shown below (Fig. 4.5).

(5-1) Start the remote transaction.
The default remote transaction isolation level is REPEATABLE READ; if the isolation level of the local transaction is set to SERIALIZABLE, the remote transaction is also set to SERIALIZABLE.

(5-2)-(5-4) Declare a cursor.
The SQL statement is basically executed as a cursor.

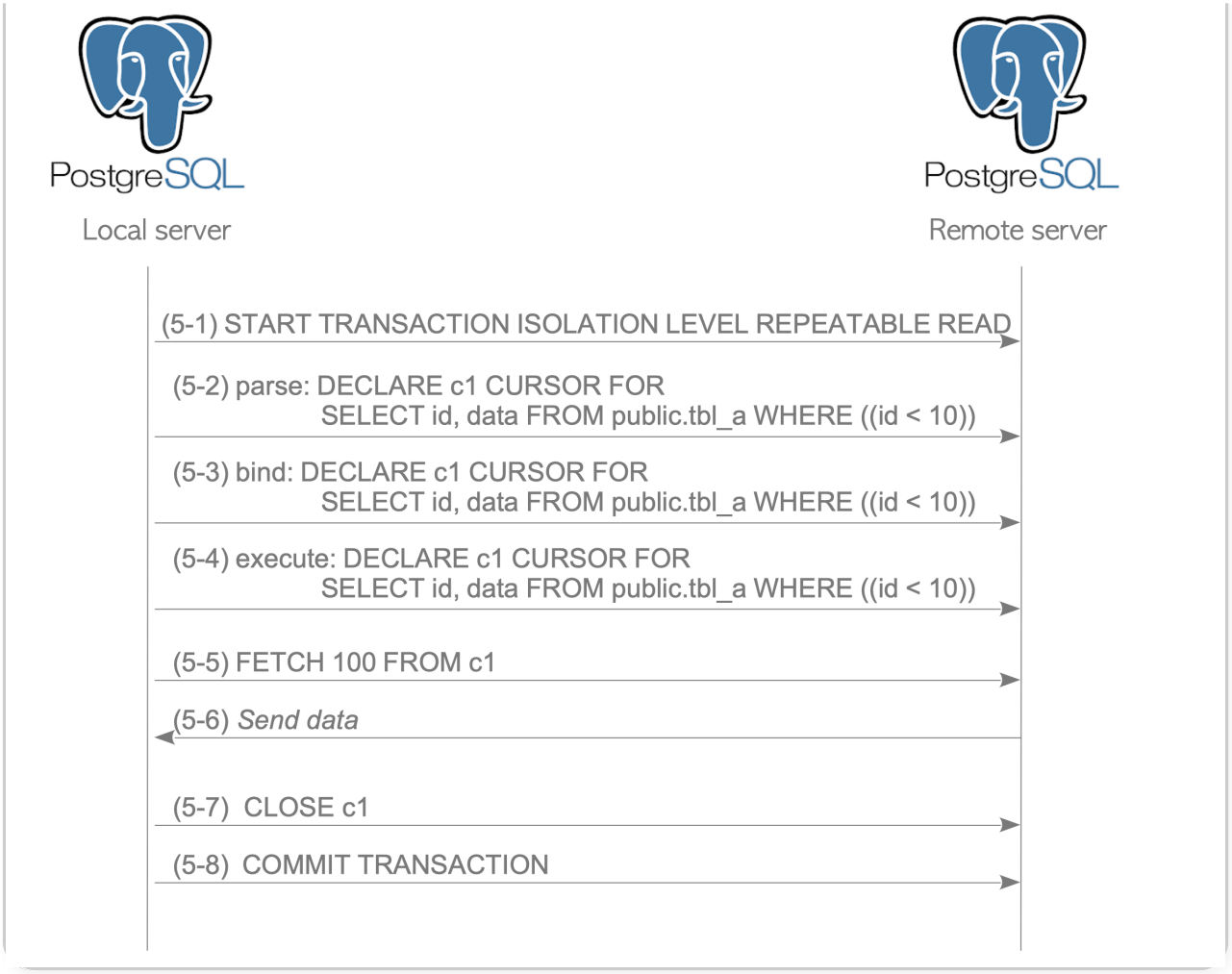
(5-5) Execute FETCH commands to obtain the result.
By default, 100 rows are fetched by the FETCH command.

(5-6) Receive the result from the remote server.

(5-7) Close the cursor.

(5-8) Commit the remote transaction.

Fig. 4.5. Typical sequence of SQL statements to execute a SELECT query in postgres_fdw.



The actual log of the remote server can be found here.

ℹ The default remote transaction isolation level in postgres_fdw.

The explanation for why the default remote transaction isolation level is REPEATABLE READ is provided in the official document.

> The remote transaction uses the SERIALIZABLE isolation level when the local transaction has

> the SERIALIZABLE isolation level; otherwise it uses the 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.

4.1.2. How the Postgres_fdw Extension Performs

postgres_fdw extension is a special module that is officially maintained by the PostgreSQL Global Development Group and its source code is included in the PostgreSQL source code tree.

postgres_fdw is gradually improved. Table 4.1 presents the release notes related to postgres_fdw

from the official document.

Table 4.1 Release notes related to postgres_fdw (cited from the official document).

Version	Description
9.3	postgres_fdw module is released.
9.6	Consider performing sorts on the remote server. Consider performing joins on the remote server. When feasible, perform UPDATE or DELETE entirely on the remote server. Allow the fetch size to be set as a server or table option.
10	Push aggregate functions to the remote server, when possible.
11	Allow to push down aggregates to foreign tables that are partitions. Allow to push UPDATEs and DELETEs using joins to foreign servers.
12	Allow ORDER BY sorts and LIMIT clauses to be pushed in more cases.

Given that the previous section describes how postgres_fdw processes a single-table query the following subsection how postgres_fdw processes a multi-table query, sort operation and aggregate functions.

This subsection focuses on the SELECT statement; however, postgres_fdw can also process other DML (INSERT, UPDATE, and DELETE) statements as shown ℹ below.

ℹ PostgreSQL's FDW does not detect deadlock.

postgres_fdw and the FDW feature do not support the distributed lock manager and the distributed deadlock detection feature. Therefore, a deadlock can be easily generated. For example, if Client_A updates a local table 'tbl_local' and a foreign table 'tbl_remote' and Client_B updates 'tbl_remote' and 'tbl_local', these two transactions are in deadlock but could not be detected by PostgreSQL. Therefore these transactions could not be committed.

```
localdb=# -- Client A
localdb=# BEGIN;
BEGIN
localdb=# UPDATE tbl_local SET data = 0 WHER
E id = 1;
```

```
UPDATE 1
localdb=# UPDATE tbl_remote SET data = 0 W
HERE id = 1;
UPDATE 1
```

```
localdb=# -- Client B
localdb=# BEGIN;
BEGIN
localdb=# UPDATE tbl_remote SET data = 0 W
HERE id = 1;
UPDATE 1
localdb=# UPDATE tbl_local SET data = 0 WHER
E id = 1;
UPDATE 1
```

4.1.2.1. Multi-Table Query

To execute a multi-table query, postgres_fdw fetches each foreign table using a single-table SELECT statement and then join them on the local server.

In version 9.5 or earlier, even if the foreign tables are stored in the same remote server, postgres_fdw fetches them individually and joins them.

In version 9.6 or later, postgres_fdw has been improved and can execute the remote join

operation on the remote server when the foreign tables are on the same server and the use_remote_estimate option is on.

The execution details are described as follows.

Version 9.5 or earlier:

Let us explore how PostgreSQL processes the following query that joins two foreign tables: *tbl_a* and *tbl_b*.

```
localdb=# SELECT * FROM tbl_a AS a, tbl_b AS
 b WHERE a.id = b.id AND a.id < 200;
```

The result of the EXPLAIN command of the query is shown below.

```
1.  localdb=# EXPLAIN SELECT * FROM tbl_a A
    S a, tbl_b AS b WHERE a.id = b.id AND a
    .id < 200;
2.                                  QUERY
    PLAN
3.  ------------------------------------------
    ------------------------------------------
4.   Merge Join  (cost=532.31..700.34 row
    s=10918 width=16)
5.     Merge Cond: (a.id = b.id)
6.     ->  Sort  (cost=200.59..202.72 rows
    =853 width=8)
```

```
7.            Sort Key: a.id
8.            ->  Foreign Scan on tbl_a a
    (cost=100.00..159.06 rows=853 width=8)
9.      ->  Sort  (cost=331.72..338.12 rows
    =2560 width=8)
10.           Sort Key: b.id
11.           ->  Foreign Scan on tbl_b b
    (cost=100.00..186.80 rows=2560 width=8)
12.   (8 rows)
```

The result shows that the executor selects the merge join and is processed as the following steps:

Line 8: The executor fetches the table tbl_a using the foreign table scan.

Line 6: The executor sorts the fetched rows of tbl_a on the local server.

Line 11: The executor fetches the table tbl_b using the foreign table scan.

Line 9: The executor sorts the fetched rows of tbl_b on the local server.

Line 4: The executor carries out a merge join operation on the local server.

The following describes how the executor fetches the rows (Fig. 4.6).

(5-1) Start the remote transaction.

(5-2) Declare the cursor c1, the SELECT statement of which is shown below:

```
SELECT id,data FROM public.tbl_a WHERE (id < 200)
```

(5-3) Execute FETCH commands to obtain the result of the cursor 1.

(5-4) Declare the cursor c2, whose SELECT statement is shown below:

```
SELECT id,data FROM public.tbl_b
```

Note that the WHERE clause of the original double-table query is "tbl_a.id = tbl_b.id AND tbl_a.id < 200"; therefore, a WHERE clause "tbl_b.id < 200" can be logically added to the SELECT statement as shown previously. However, postgres_fdw cannot perform this inference; therefore, the executor has to execute the SELECT statement that does not contain any WHERE clauses and has to fetch all rows of the foreign table *tbl_b*.

This process is inefficient because unnecessary rows must be read from the remote server via the network. Furthermore, the received rows must be sorted to execute the merge join.

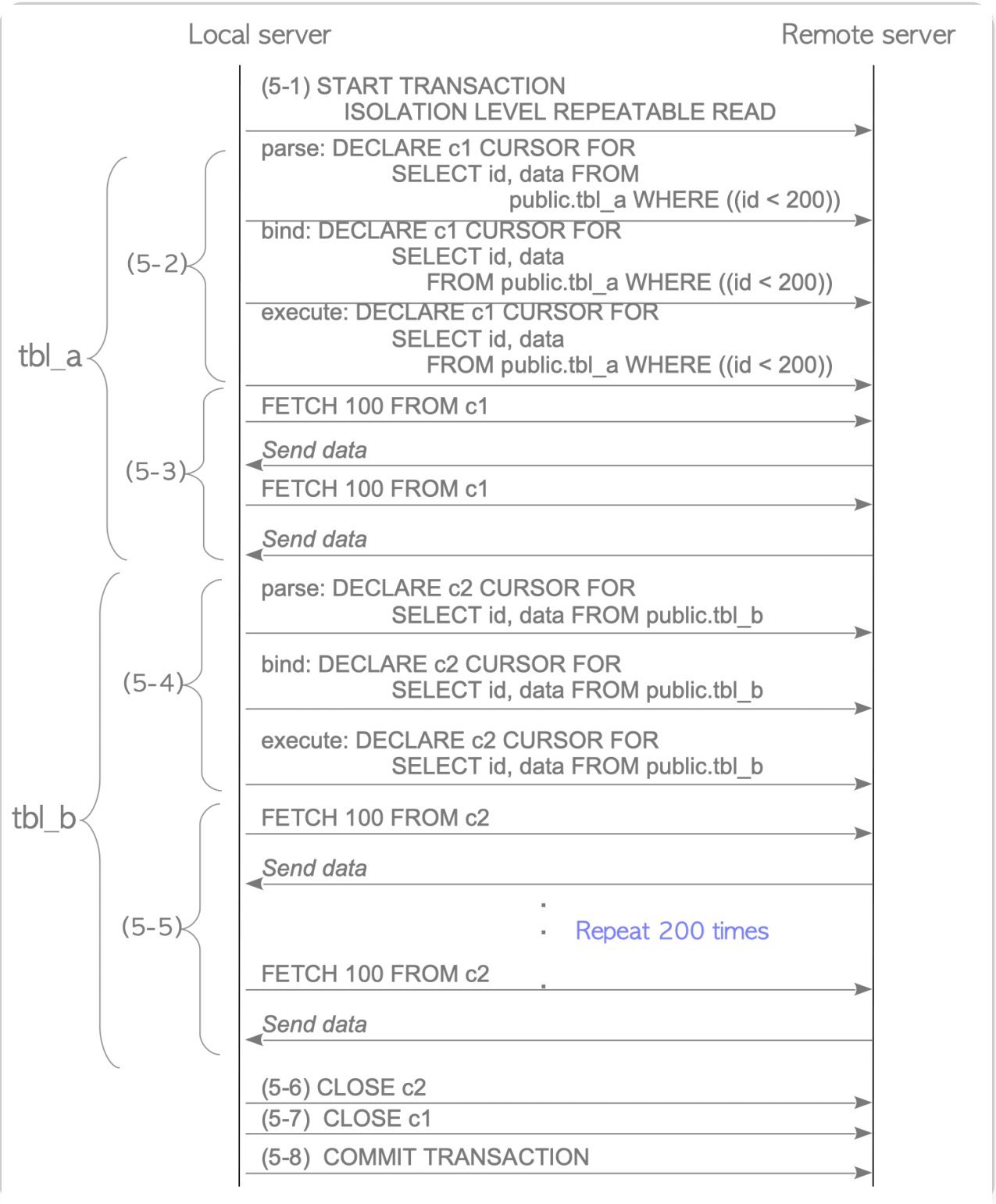
(5-5) Execute FETCH commands to obtain the result of the cursor 2.

(5-6) Close the cursor c1.

(5-7) Close the cursor c2.

(5-8) Commit the transaction.

Fig. 4.6. Sequence of SQL statements to execute the Multi-Table Query in version 9.5 or earlier.



The actual log of the remote server can be found here.

After receiving the rows, the executor sorts both received rows of tbl_a and tbl_b, and then executes a merge join operation with the sorted rows.

Version 9.6 or later:

If the use_remote_estimate option is *on* (the default is off), postgres_fdw sends several EXPLAIN commands to obtain the costs of all plans related to the foreign tables.

To send the EXPLAIN commands, postgres_fdw sends both the EXPLAIN command of each single-table query and the EXPLAIN commands of the SELECT statements to execute remote join operations. In this example, the following seven EXPLAIN commands are sent to the remote server to obtain the estimated costs of each SELECT statement; the planner then selects the cheapest plan.

```
(1) EXPLAIN SELECT id, data FROM public.tbl_a
WHERE ((id < 200))
(2) EXPLAIN SELECT id, data FROM public.tbl_b
(3) EXPLAIN SELECT id, data FROM public.tbl_a
WHERE ((id < 200)) ORDER BY id ASC NULLS LAST
(4) EXPLAIN SELECT id, data FROM public.tbl_a
```

```
WHERE ((((SELECT null::integer)::integer) = i
d)) AND ((id < 200))
(5) EXPLAIN SELECT id, data FROM public.tbl_b
ORDER BY id ASC NULLS LAST
(6) EXPLAIN SELECT id, data FROM public.tbl_b
WHERE ((((SELECT null::integer)::integer) = i
d))
(7) EXPLAIN SELECT r1.id, r1.data, r2.id, r2.
data FROM (public.tbl_a r1 INNER JOIN public
.tbl_b r2 ON (((r1.id = r2.id)) AND ((r1.id <
200))))
```

Let us execute the EXPLAIN command on the local server to observe what plan is selected by the planner.

```
1. localdb=# EXPLAIN SELECT * FROM tbl_a A
   S a, tbl_b AS b WHERE a.id = b.id AND a
   .id < 200;
2.                              QUERY PLAN
3. ------------------------------------------
   ----------------------
4.  Foreign Scan  (cost=134.35..244.45 r
   ows=80 width=16)
5.    Relations: (public.tbl_a a) INNER
    JOIN (public.tbl_b b)
6. (2 rows)
```

The result shows that the planner selects the inner join query that is processed on the remote

server, which is very efficient.

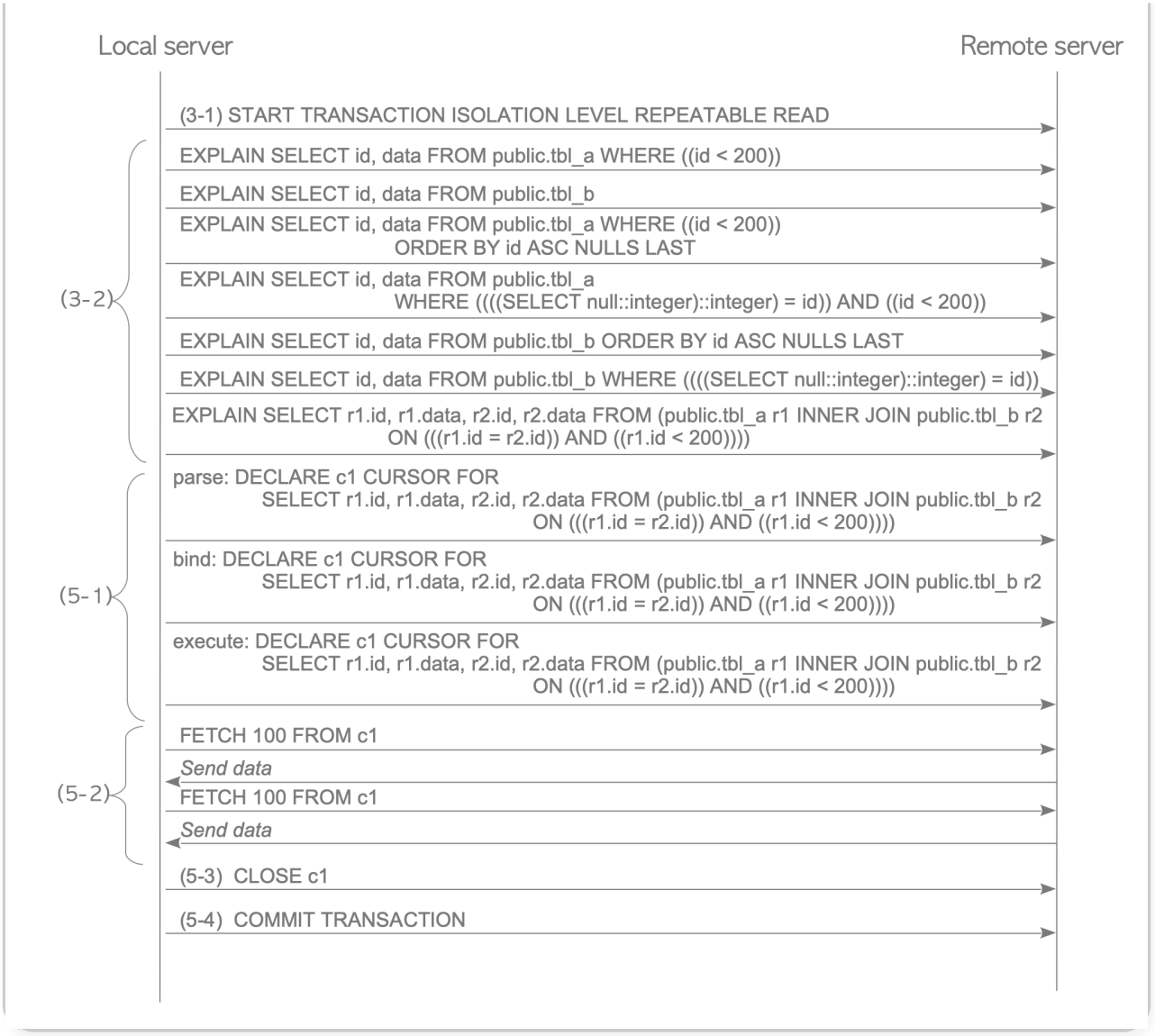
The following describes how postgres_fdw is performed (Fig. 4.7).

(3-1) Start the remote transaction.
(3-2) Execute the EXPLAIN commands for estimating the cost of each plan path.
In this example, seven EXPLAIN commands are executed. Then, the planner selects the cheapest cost of the SELECT queries using the results of the executed EXPLAIN commands.
(5-1) Declare the cursor c1, whose SELECT statement is shown below:

```
SELECT r1.id, r1.data, r2.id, r2.data FROM
(public.tbl_a r1 INNER JOIN public.tbl_b
r2
  ON (((r1.id = r2.id)) AND ((r1.id < 200
))))
```

(5-2) Receive the result from the remote server.
(5-3) Close the cursor c1.
(5-4) Commit the transaction.

Fig. 4.7. Sequence of SQL statements to execute the remote-join operation in version 9.6 or later.



The actual log of the remote server can be found here.

Note that if the use_remote_estimate option is off (by default), a remote-join query is rarely selected because the costs are estimated using a very large embedded value.

4.1.2.2. Sort Operations

In version 9.5 or earlier, the sort operation, such as ORDER BY, is processed on the local server, i.e.

the local server fetches all the target rows from the remote server prior to the sort operation. Let us explore how a simple query that includes an ORDER BY clause is processed using the EXPLAIN command.

```
1. localdb=# EXPLAIN SELECT * FROM tbl_a A
   S a WHERE a.id < 200 ORDER BY a.id;
2.                                QUERY PLA
   N
3. ------------------------------------------
   ------------------------------------
4.  Sort  (cost=200.59..202.72 rows=853 w
   idth=8)
5.    Sort Key: id
6.    ->  Foreign Scan on tbl_a a  (cost
   =100.00..159.06 rows=853 width=8)
7. (3 rows)
```

Line 6: The executor sends the following query to the remote server, and then fetches the query result.

```
SELECT id, data FROM public.tbl_a WHERE
((id < 200))
```

Line 4: The executor sorts the fetched rows of tbl_a on the local server.

The actual log of the remote server can be found here.

In version 9.6 or later, postgres_fdw can execute the SELECT statements with an ORDER BY clause on the remote server if possible.

```
1. localdb=# EXPLAIN SELECT * FROM tbl_a A
   S a WHERE a.id < 200 ORDER BY a.id;
2.                                QUERY PLAN
3. ------------------------------------------
   ------------------------------
4.  Foreign Scan on tbl_a a  (cost=100.0
   0..167.46 rows=853 width=8)
5. (1 row)
```

Line 4: The executor sends the following query that has an ORDER BY clause to the remote server and then fetches the query result, which is already sorted.

```
SELECT id, data FROM public.tbl_a WHERE
((id < 200)) ORDER BY id ASC NULLS LAST
```

The actual log of the remote server can be found here. The workload of the local server has been reduced by this improvement.

4.1.2.3. Aggregate Functions

In version 9.6 or earlier, similar to the sort operation mentioned in the previous subsection, the aggregate functions such as AVG() and cont()

are processed on the local server as the following steps.

```
1. localdb=# EXPLAIN SELECT AVG(data) FROM
   tbl_a AS a WHERE a.id < 200;
2.                                QUERY PLA
   N
3. ------------------------------------------
   ------------------------------------
4.  Aggregate  (cost=168.50..168.51 rows
   =1 width=4)
5.    ->  Foreign Scan on tbl_a a  (cost
   =100.00..166.06 rows=975 width=4)
6. (2 rows)
```

Line 5: The executor sends the following query to the remote server, and then fetches the query result.

```
SELECT id, data FROM public.tbl_a WHERE
((id < 200))
```

Line 4: The executor calculates the average of the fetched rows of tbl_a on the local server.

The actual log of the remote server can be found here. This process is costly because sending a large number of rows consumes heavy network traffic and takes a long time.

In version 10 or later, postgres_fdw executes the SELECT statement with the aggregate function on the remote server if possible.

```
1. localdb=# EXPLAIN SELECT AVG(data) FROM
   tbl_a AS a WHERE a.id < 200;
2.                          QUERY PLAN
3. ------------------------------------------------
   ---------------
4.  Foreign Scan  (cost=102.44..149.03 r
   ows=1 width=32)
5.    Relations: Aggregate on (public.
   tbl_a a)
6. (2 rows)
```

Line 4: The executor sends the following query that contains an AVG() function to the remote server, and then fetches the query result.

```
SELECT avg(data) FROM public.tbl_a WHERE
((id < 200))
```

The actual log of the remote server can be found here. This process is obviously efficient because the remote server calculates the average and sends only one row as the result.

Push-down

Similar to the given example, the **push-down** is an operation where the local server allowed the remote server to process some operations, such as aggregate procedures.

4.2. Parallel Query