

Внешние данные Обзор



Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Назначение оберток сторонних данных

Настройка доступа к внешним данным

Примеры: `postgres_fdw` и `file_fdw`

Другие доступные обертки

Представление внешних данных как обычных таблиц

стандарт ISO/IEC 9075-9 (SQL/MED)

поддерживаются

SELECT, INSERT, UPDATE, DELETE, TRUNCATE, COPY

триггеры

обычное разграничение доступа (GRANT, REVOKE)

Способ миграции данных из других СУБД

Механизм для шардинга

вместе с секционированием: Postgres Pro Shardman

В стандарте ISO/IEC 9075-9 (SQL/MED) определяется, как базы данных SQL должны работать с внешними источниками с учетом доступа как на чтение, так и на запись.

Внешние данные представлены в PostgreSQL как таблицы, которые называются *внешними* или *сторонними*. С ними можно работать при помощи обычных команд DML: INSERT, UPDATE, DELETE, SELECT, TRUNCATE, а также COPY. На внешние таблицы можно создавать триггеры, а для разграничения прав используются команды GRANT и REVOKE.

Основное отличие внешних таблиц от обычных в том, что данные не хранятся в самой БД PostgreSQL, а загружаются из внешней системы (отправляются во внешнюю систему) при выполнении запросов.

Обертки сторонних данных могут быть весьма полезны как при одновременной работе с данными из разных источников, так и при миграции данных в PostgreSQL из других СУБД.

Поддержка внешних данных активно развивается, поскольку (вместе с секционированием) является одним из компонентов для реализации шардинга: секции основной таблицы представляются внешними таблицами, физически находящимися на отдельных серверах.

<https://postgrespro.ru/docs/postgresql/16/ddl-foreign-data>

Данные из сторонних таблиц можно сохранять при логическом резервировании (используя ключ `--include-foreign-data` утилиты `pg_dump`), однако при восстановлении из такой копии нужно учитывать доступность внешних серверов и их возможности по записи данных.

<https://postgrespro.ru/docs/postgresql/16/app-pgdump>

1. Обертка сторонних данных

каков тип внешнего источника данных?

postgres_fdw: внешний сервер PostgreSQL

Чтобы настроить доступ к внешним данным, нужно создать несколько объектов базы данных.

Все начинается с *обертки сторонних данных* (foreign data wrapper, FDW). Обертка определяет тип внешнего источника данных. Например, postgres_fdw реализует доступ к базам данных PostgreSQL, а file_fdw — доступ к файлам операционной системы.

<https://postgrespro.ru/docs/postgresql/16/postgres-fdw>

<https://postgrespro.ru/docs/postgresql/16/sql-createforeigndatawrapper>

Обертка сторонних данных

```
=> CREATE DATABASE fdw_overview;
```

CREATE DATABASE

```
=> \c fdw_overview
```

You are now connected to database "fdw_overview" as user "student".

Обертка создается при создании расширения:

```
=> CREATE EXTENSION postgres_fdw;
```

CREATE EXTENSION

```
=> \dew
```

List of foreign-data wrappers			
Name	Owner	Handler	Validator
postgres_fdw	student	postgres_fdw_handler	postgres_fdw_validator
(1 row)			

1. Обертка сторонних данных

каков тип внешнего источника данных?

2. Внешний сервер

как подключиться к внешнему источнику?

postgres_fdw: узел, порт, база данных

Следующий объект — *внешний сервер*. Сервер определяет, как подключаться к внешнему источнику данных.

Например, если источником является СУБД, то надо знать узел, порт, имя базы данных. (Но не имя пользователя и пароль. Эта информация появится чуть позже.)

Если требуется подключение к нескольким внешним источникам (одного типа, который определен оберткой), то для каждого из них нужно создать отдельный сервер.

<https://postgrespro.ru/docs/postgresql/16/sql-createserver>

Внешний сервер

В качестве внешнего сервера выберем базу данных книжного магазина на нашем же локальном сервере.

```
=> CREATE SERVER remote_server
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (
    host 'localhost',
    port '5432',
    dbname 'bookstore2'
);
```

CREATE SERVER

Обратите внимание, что в параметрах сервера не указывается роль.

```
=> \x \des+ \x
```

Expanded display is on.

List of foreign servers

```
-[ RECORD 1 ]-----+-----
Name          | remote_server
Owner         | student
Foreign-data wrapper | postgres_fdw
Access privileges |
Type          |
Version       |
FDW options   | (host 'localhost', port '5432', dbname 'bookstore2')
Description   |
```

Expanded display is off.

1. Обертка сторонних данных

каков тип внешнего источника данных?

2. Внешний сервер

как подключиться к внешнему источнику?

3. Сопоставление ролей

как локальные роли связаны с разграничением доступа, используемым внешним источником?

postgres_fdw: и там, и там — обычные роли PostgreSQL

Далее следует сопоставление ролей.

Внешний источник может выполнять аутентификацию при подключении. В таких случаях необходимо определить, какие локальные роли могут подключаться к каким учетным записям на внешнем сервере.

Если используется аутентификация по паролю, то здесь же указывается и пароль.

Суперпользователь может разрешить обычным пользователям использовать подключение без пароля, установив параметр `password_required` в значение `false`.

<https://postgrespro.ru/docs/postgresql/16/sql-createusermapping>

Сопоставление ролей

Настроим соответствие: пусть локальная роль student подключается к внешнему серверу как postgres.

```
=> CREATE USER MAPPING FOR student
SERVER remote_server
OPTIONS (
    user 'postgres',
    password 'postgres'
);
```

CREATE USER MAPPING

```
=> \deu+
```

List of user mappings		
Server	User name	FDW options
remote_server	student	("user" 'postgres', password 'postgres')
(1 row)		

Разумеется, можно настраивать несколько соответствий для разных ролей.

1. Обертка сторонних данных

каков тип внешнего источника данных?

2. Внешний сервер

как подключиться к внешнему источнику?

3. Сопоставление ролей

как локальные роли связаны с ролями внешнего источника?

4. Внешние таблицы

какова структура внешних данных?

postgres_fdw: данные в обычных таблицах

И, наконец, можно приступить к созданию *внешних таблиц*. Внешняя таблица определяет, как представить данные из внешнего источника в виде обычной реляционной таблицы.

Для внешних таблиц можно добавлять ограничения целостности NOT NULL и CHECK. Эти ограничения не будут применяться при выполнении команд, ведь в любом случае PostgreSQL не сможет обеспечить соблюдение ограничений на внешнем источнике.

Но планировщик может использовать информацию об ограничениях при построении плана выполнения запроса.

Обертка postgres_fdw для выборки данных использует пакетный режим, записи передаются пакетами, размер которых задается параметром сервера fetch_size (по умолчанию 100 строк). Вставка (с помощью команд INSERT и COPY FROM) также использует пакетный режим, за количество строк здесь отвечает параметр сервера batch_size. Для изменения и удаления строк пакетный режим пока не реализован.

<https://postgrespro.ru/docs/postgresql/16/sql-createforeigntable>

<https://postgrespro.ru/docs/postgresql/16/sql-importforeignschema>

Внешние таблицы

Внешнюю таблицу можно создать явным образом, при необходимости указывая названия объектов, если они отличаются:

```
=> CREATE FOREIGN TABLE remote_users (  
    id integer OPTIONS (column_name 'user_id') NOT NULL,  
    username text NOT NULL,  
    email text NOT NULL  
)  
SERVER remote_server  
OPTIONS (  
    schema_name 'public',  
    table_name 'users'  
);
```

CREATE FOREIGN TABLE

Можно указать только некоторые ограничения целостности, но в любом случае они не проверяются локально и просто отражают ограничения, накладываемые внешней системой.

```
=> \det
```

```
          List of foreign tables  
Schema |      Table      |      Server  
-----+-----+-----  
public | remote_users | remote_server  
(1 row)
```

```
=> SELECT * FROM remote_users WHERE id = 1;
```

```
id | username |      email  
---+-----+-----  
1 | alice   | alice@localhost  
(1 row)
```

Как выполняется такой запрос? Иными словами, передается ли предикат внешнему серверу, чтобы он сам выбрал эффективный способ выполнения, или фильтрацией занимается локальный сервер?

```
=> EXPLAIN (verbose, costs off)  
    SELECT * FROM remote_users WHERE id = 1;
```

QUERY PLAN

```
-----  
Foreign Scan on public.remote_users  
Output: id, username, email  
Remote SQL: SELECT user_id, username, email FROM public.users WHERE ((user_id = 1))  
(3 rows)
```

Планировщик умеет распределять работу между серверами. Также ничто не мешает использовать в одном запросе как внешние, так и локальные данные.

```
=> EXPLAIN (costs off)  
    SELECT username FROM remote_users  
    UNION ALL  
    SELECT rolname FROM pg_roles;
```

QUERY PLAN

```
-----  
Append  
-> Subquery Scan on "*SELECT* 1"  
    -> Foreign Scan on remote_users  
-> Subquery Scan on "*SELECT* 2"  
    -> Seq Scan on pg_authid  
(5 rows)
```

Другой способ — не создавать внешние таблицы по одной, а импортировать внешнюю схему (всю или выборочно несколько таблиц), если обертка это позволяет.

```
=> CREATE SCHEMA bookstore2;
```

CREATE SCHEMA

```
=> IMPORT FOREIGN SCHEMA public
    LIMIT TO (users, sessions)
    FROM SERVER remote_server
    INTO bookstore2;
```

IMPORT FOREIGN SCHEMA

```
=> SELECT * FROM bookstore2.users;
```

user_id	username	email
1	alice	alice@localhost
2	bob	bob@localhost

(2 rows)

Обертка postgres_fdw позволяет и изменять данные:

```
=> EXPLAIN (analyze, verbose, costs off)
    UPDATE remote_users
    SET email = 'alice@gmail.com'
    WHERE id = 1;
```

QUERY PLAN

```
-----
Update on public.remote_users (actual time=0.580..0.580 rows=0 loops=1)
-> Foreign Update on public.remote_users (actual time=0.579..0.579 rows=1 loops=1)
    Remote SQL: UPDATE public.users SET email = 'alice@gmail.com'::text WHERE
((user_id = 1))
Planning Time: 0.095 ms
Execution Time: 0.743 ms
(5 rows)
```

Удаленное соединение

открывается при первом обращении к внешним данным
по умолчанию остается открытым до конца сеанса
автоматически восстанавливается при разрыве

Удаленная транзакция

фиксируется или прерывается автоматически, когда фиксируется
или прерывается локальная транзакция
Repeatable Read для локальных транзакций уровней Read Committed
и Repeatable Read
Serializable для локальных транзакций Serializable
согласованность не гарантируется (нет двухфазной фиксации)

При работе с другой базой данных через `postgres_fdw` важно понимать, как происходит управление соединениями и транзакциями.

По умолчанию обертка `postgres_fdw` автоматически открывает соединение при первом обращении к внешнему серверу (одно на каждую роль) и оставляет его открытым до конца локального сеанса. Если обертка обнаружит, что соединение разорвано, она автоматически попытается установить новое подключение.

Удаленные транзакции «привязаны» к локальным:

- расширение автоматически начинает удаленную транзакцию, когда локальная обращается к внешним данным;
- удаленная транзакция автоматически завершается с завершением локальной (фиксацией или обрывом).

Тем не менее, локальная и удаленная транзакции — это *разные* транзакции, а не одна глобальная. Расширение `postgres_fdw` пока не позволяет использовать двухфазную фиксацию, поэтому согласованность не гарантируется. Например, локальная транзакция может быть зафиксирована, а удаленная — нет из-за сетевого сбоя.

При фиксации транзакции, в которой участвуют несколько внешних серверов, подключенных через `postgres_fdw`, удаленные транзакции могут завершаться асинхронно, а не последовательно одна за другой.

<https://postgrespro.ru/docs/postgresql/16/postgres-fdw#POSTGRES-FDW-CONNECTION-MANAGEMENT>

Соединения и транзакции

Когда выполнялся запрос к внешней таблице `remote_users`, обертка открыла соединение и по умолчанию не закрывает его:

```
=> SELECT datname, pid FROM pg_stat_activity
WHERE application_name = 'postgres_fdw';
```

```
   datname   | pid
-----+-----
 bookstore2 | 46381
(1 row)
```

Завершим обслуживающий процесс...

```
=> SELECT pg_terminate_backend(46381);
```

```
pg_terminate_backend
-----
t
(1 row)
```

...и еще раз обратимся к внешней таблице:

```
=> SELECT * FROM remote_users;
```

```
 id | username |      email
----+-----+-----
  2 | bob      | bob@localhost
  1 | alice    | alice@gmail.com
(2 rows)
```

Нам удалось получить данные! Обертка автоматически восстановила подключение, соединение теперь обслуживается другим процессом:

```
=> SELECT datname, pid FROM pg_stat_activity
WHERE application_name = 'postgres_fdw';
```

```
   datname   | pid
-----+-----
 bookstore2 | 46766
(1 row)
```

Поведение можно изменить, задав атрибут внешнего сервера

```
keep_connections 'off'
```

Тогда обертка будет открывать соединение при каждом обращении к внешней таблице и закрывать его после обращения.

Сравнение с dblink



	postgres_fdw	dblink
доступные команды SQL	SELECT, INSERT, UPDATE, DELETE, TRUNCATE, COPY	любые
совмещение в запросе локальных и внешних данных	да	сложно
управление соединением	автоматическое	ручное
управление транзакциями	автоматическое	ручное
глобальные транзакции	нет	нет
стандартизированный способ	да	нет

14

В теме «Фоновые процессы» рассматривалось расширение dblink, которое предназначено для выполнения запросов на удаленном сервере PostgreSQL и предоставляет возможности, схожие с возможностями postgres_fdw.

Обертка postgres_fdw позволяет выполнять ограниченный набор команд SQL, но в большинстве случаев более удобна: она сама управляет соединениями и транзакциями («привязывая» их к локальному соединению и транзакции). Обертка позволяет использовать в одном запросе как локальные, так и внешние данные; при этом автоматически формируется распределенный план выполнения запроса. Синтаксис доступа к данным определяется стандартом SQL.

Зато расширение dblink позволяет выполнять любые команды SQL. В некоторых случаях может быть нужным управлять соединениями и транзакциями вручную.

Однако оба способа не предоставляют глобальных (распределенных) транзакций.

Таким образом, для доступа к внешним данным PostgreSQL в большинстве случаев следует выбирать postgres_fdw. Расширение dblink стоит использовать, если нужны возможности, которые не предоставляет обертка сторонних данных.

1. Обертка сторонних данных

тип внешнего источника — текстовый файл с разделителями

2. Внешний сервер

как подключиться — не требуется дополнительной информации

3. Сопоставление ролей

не имеет смысла, т. к. во внешнем источнике нет ролей

4. Внешние таблицы

структура данных — описание полей файла в виде столбцов таблицы

Еще одну обертку сторонних данных предоставляет расширение file_fdw, предназначенное для доступа к файлам в файловой системе сервера PostgreSQL.

Доступ возможен только на чтение и выполняется с помощью того же механизма, что и SQL-команда COPY.

<https://postgrespro.ru/docs/postgresql/16/file-fdw>

Расширение file_fdw

Расширение file_fdw позволяет обращаться к любому текстовому файлу и задействует тот же механизм, что и команда COPY. Поэтому для демонстрации мы выгрузим строки из имеющейся таблицы в файл, а затем прочитаем данные из файла.

```
=> COPY (SELECT * FROM remote_users)
TO '/var/lib/postgresql/16/users.txt'
WITH (
    format 'text',
    delimiter '/'
);
```

COPY 2

```
student$ cat /var/lib/postgresql/16/users.txt
```

```
2/bob/bob@localhost
1/alice/alice@gmail.com
```

```
=> CREATE EXTENSION file_fdw;
```

CREATE EXTENSION

```
=> CREATE SERVER file_server
    FOREIGN DATA WRAPPER file_fdw;
```

CREATE SERVER

```
=> CREATE FOREIGN TABLE file_users (
    id integer,
    username text,
    email text
)
SERVER file_server
OPTIONS (
    filename '/var/lib/postgresql/16/users.txt',
    format 'text',
    delimiter '/'
);
```

CREATE FOREIGN TABLE

```
=> SELECT * FROM file_users;
```

id	username	email
2	bob	bob@localhost
1	alice	alice@gmail.com

(2 rows)

Базы данных

ODBC, JDBC

различные реляционные и NoSQL СУБД

Файлы различных форматов

Геоданные и научные данные

Веб-ресурсы

стандартные протоколы (RSS, IMAP, WWW...)

для различных сайтов

Можно написать собственную обертку

на C или Python (с помощью расширения Multicorn)

Рассмотренные обертки сторонних данных, `file_fdw` и `postgres_fdw`, поставляются вместе с PostgreSQL в виде contrib-модулей.

Помимо этого существует множество других оберток, которые предоставляют доступ ко многим популярным базам данных и не только. Информацию о них можно получить:

- на страничке wiki: <https://wiki.postgresql.org/wiki/Fdw>
- на сайте PGXN: <http://pgxn.org/tag/fdw/>

Если нужную обертку не удалось найти, то ее можно написать самостоятельно. Для этого потребуется язык C:

<https://postgrespro.ru/docs/postgresql/16/fdwhandler>

Другой вариант — воспользоваться расширением Multicorn (<https://multicorn.org/>), которое позволяет написать обертку на языке Python.

Обертки сторонних данных — стандартный способ работы с данными любых внешних источников

доступ к PostgreSQL и файлам «из коробки»

есть реализации оберток для огромного числа систем

Альтернатива — расширение `dblink` для выполнения произвольных запросов на удаленном сервере PostgreSQL



1. Поставщик информирует магазин о новинках книжного рынка, предоставляя CSV-файл определенного формата. Загрузите информацию из файла `new_books.csv`, расположенного в домашнем каталоге пользователя `student`, в таблицы базы данных. Формат файла описан в комментариях к слайду.
Учтите, что авторы, указанные в файле, могут уже существовать в базе данных, но в файле могут быть допущены опечатки.
2. Проверьте в приложении, что новые книги успешно переданы на логическую реплику, и для них работает полнотекстовый поиск.

1. CSV-файл содержит следующие поля:

- фамилия автора;
- имя автора;
- отчество автора;
- название книги;
- формат издания;
- количество страниц;
- ISBN;
- аннотация;
- издательство;
- год выпуска;
- гарнитура;
- номер издания;
- серия;
- имя файла с обложкой (файл находится в каталоге `covers`).

Если у книги несколько авторов, то они следуют в файле в том порядке, в котором указаны в выходных сведениях. При этом для второго и следующих авторов все поля, относящиеся к книге — пустые.

Воспользуйтесь расширением `file_fdw`, указав для файла формат CSV.

Для борьбы с опечатками вспомните про расширение `pg_trgm`, с которым вы познакомились в практике к теме «Классы операторов».

2. Логическая реплика с подпиской на изменения книг настраивалась в задании к теме «Логическая репликация».

1. Загрузка новых поступлений

```
=> CREATE EXTENSION file_fdw;

CREATE EXTENSION

=> CREATE SERVER file_server
    FOREIGN DATA WRAPPER file_fdw;

CREATE SERVER

=> CREATE FOREIGN TABLE new_books (
    last_name text,
    first_name text,
    middle_name text,
    title text,
    format text,
    pages integer,
    isbn text,
    abstract text,
    publisher text,
    year text,
    typeface text,
    edition integer,
    series text,
    cover_filename text
)
SERVER file_server
OPTIONS (
    filename '/home/student/new_books.csv',
    format 'csv'
);

CREATE FOREIGN TABLE
```

Здесь предполагается, что несколько процедур загрузки книг не будут запускаться параллельно. Если такой гарантии нет, в коде надо отдельно предусмотреть эту возможность, устанавливая необходимые блокировки.

Для сопоставления авторов используется нечеткий поиск с помощью триграмм. Оператор % отсекает совсем непохожих (предел похожести по умолчанию 0.3 слишком низок для нашего случая, поэтому увеличиваем его). Из оставшихся выбираем лучшее совпадение, упорядочивая кандидатов с помощью функции similarity.

```
=> CREATE EXTENSION pg_trgm;

CREATE EXTENSION

=> SELECT set_limit(0.7); -- предел похожести для оператора %

set_limit
-----
      0.7
(1 row)
```

```

=> DO $$
DECLARE
    i new_books;
    book_id bigint;
    author_id bigint;
    author_cause text;
    seq_num integer;
BEGIN
    FOR i IN SELECT * FROM new_books
    LOOP
        IF i.title IS NOT NULL THEN
            INSERT INTO books AS b(
                title, format, pages, additional, cover
            ) VALUES (
                i.title,
                i.format::text::book_format,
                i.pages,
                jsonb_build_object(
                    'ISBN', i.isbn,
                    'Аннотация', i.abstract,
                    'Издательство', i.publisher,
                    'Год выпуска', i.year,
                    'Гарнитура', i.typeface,
                    'Издание', i.edition,
                    'Серия', i.series
                ),
                pg_read_binary_file(
                    '/home/student/covers/' || i.cover_filename
                )
            )
            RETURNING b.book_id INTO book_id;
            seq_num := 1;
            RAISE NOTICE 'Книга: % (%)', i.title, book_id;
        END IF;

        -- есть ли точное совпадение с имеющимся автором?
        SELECT a.author_id, 'точное совпадение'
        INTO author_id, author_cause
        FROM authors a
        WHERE a.last_name = i.last_name
            AND a.first_name = i.first_name
            AND a.middle_name = i.middle_name;
        -- если нет, то может найдется очень похожий?
        IF author_id IS NULL THEN
            SELECT a.author_id, 'Неточное совпадение'
            INTO author_id, author_cause
            FROM authors a
            WHERE (a.last_name || a.first_name || a.middle_name) %
                (i.last_name || i.first_name || i.middle_name)
            ORDER BY similarity(
                (a.last_name || a.first_name || a.middle_name),
                (i.last_name || i.first_name || i.middle_name)
            ) DESC
            LIMIT 1;
        END IF;
        -- если и похожего нет, то считаем автора новым
        IF author_id IS NULL THEN
            INSERT INTO authors AS a(first_name, last_name, middle_name)
            VALUES (i.first_name, i.last_name, i.middle_name)
            RETURNING a.author_id, 'новый'
            INTO author_id, author_cause;
        END IF;
        RAISE NOTICE 'Автор: %, % (%)',
            i.last_name, author_cause, author_id;

        INSERT INTO authorships(book_id, author_id, seq_num)
        VALUES (book_id, author_id, seq_num);
        seq_num := seq_num + 1;
    END LOOP;
END;
$$;

```

NOTICE: Книга: PostgreSQL 16 изнутри (101)
NOTICE: Автор: Рогов, новый (162)
NOTICE: Книга: PostgreSQL 15 изнутри (102)
NOTICE: Автор: Рогов, точное совпадение (162)
NOTICE: Книга: Путеводитель по базам данных (103)
NOTICE: Автор: Комаров, новый (163)
NOTICE: Книга: PostGIS в действии (104)
NOTICE: Автор: Обе, новый (164)
NOTICE: Автор: Хсу, новый (165)
NOTICE: Книга: Жемчужины разработки. Чему мы научились за 50 лет создания ПО (105)
NOTICE: Автор: Вигерс, новый (166)
NOTICE: Книга: Мониторинг PostgreSQL (106)
NOTICE: Автор: Лесовский, новый (167)
NOTICE: Книга: Думай о смысле. Будни переводчика IT-текстов (107)
NOTICE: Автор: Чаплыгин, новый (168)
NOTICE: Книга: Идеи, определившие облик информатики (108)
NOTICE: Автор: Льюис, новый (169)
DO

=> DROP FOREIGN TABLE new_books;

DROP FOREIGN TABLE

1. В двух разных базах данных находятся две таблицы с одинаковым набором столбцов. Выведите строки, которые присутствуют в первой таблице, но отсутствуют во второй. Решите задачу с помощью `postgres_fdw` и с помощью `dblink` и сравните два способа.
2. Как можно проверить, какой уровень изоляции использует обертка `postgres_fdw` и как она управляет соединениями и транзакциями?

1. Расширение `dblink` демонстрировалось в теме «Фоновые процессы».

2. Настройте на втором сервере журнал сообщений таким образом, чтобы в него записывалась информация:

- о подключениях (*log_connections*);
- отключениях (*log_disconnections*);
- выполняемых командах (*log_statement*).

Обратитесь к таблице на втором сервере с помощью `postgres_fdw` и проверьте, что попало в журнал сообщений.

1. Сравнение таблиц в разных базах

Таблица в первой базе данных:

```
=> CREATE DATABASE fdw_overview;
```

CREATE DATABASE

```
=> \c fdw_overview
```

You are now connected to database "fdw_overview" as user "student".

```
=> CREATE TABLE test (  
    s text  
);
```

CREATE TABLE

```
=> INSERT INTO test VALUES ('foo'),('bar'),('baz');
```

INSERT 0 3

И во второй (расположим ее на втором сервере в БД postgres):

```
student$ psql -p 5433 -U postgres -d postgres
```

```
| => CREATE TABLE test (  
|     s text  
| );  
|  
| CREATE TABLE  
|  
| => INSERT INTO test VALUES ('foo'),('bar');  
|  
| INSERT 0 2
```

Создаем стороннюю таблицу для доступа к таблице во второй базе данных:

```
=> CREATE EXTENSION postgres_fdw;
```

CREATE EXTENSION

```
=> CREATE SERVER remote_server  
FOREIGN DATA WRAPPER postgres_fdw  
OPTIONS (  
    host 'localhost',  
    port '5433',  
    dbname 'postgres'  
);
```

CREATE SERVER

На втором сервере используем роль postgres:

```
=> CREATE USER MAPPING FOR student  
SERVER remote_server  
OPTIONS (  
    user 'postgres' -- пароль не нужен, аутентификация trust  
);
```

CREATE USER MAPPING

```
=> CREATE FOREIGN TABLE test2 (  
    s text  
)  
SERVER remote_server  
OPTIONS (  
    schema_name 'public',  
    table_name 'test'  
);
```

CREATE FOREIGN TABLE

Сравниваем:

```
=> SELECT * FROM test  
EXCEPT  
SELECT * FROM test2;
```

```
s
-----
baz
(1 row)
```

С помощью расширения dblink задачу можно решить следующим образом:

```
=> CREATE EXTENSION dblink;

CREATE EXTENSION

=> SELECT * FROM test
EXCEPT
SELECT * FROM dblink(
    'host=localhost port=5433 dbname=postgres user=postgres',
    'SELECT * FROM test'
) AS (s text);

s
-----
baz
(1 row)
```

В этом случае подготовительные действия практически не требуются, но сам запрос выглядит сложнее.

2. Проверка работы postgres_fdw

На втором сервере настроим журнал сообщений так, чтобы в него попадала информация о подключениях и отключениях, а также о выполняемых командах.

```
=> ALTER SYSTEM SET log_connections = on;

ALTER SYSTEM

=> ALTER SYSTEM SET log_disconnections = on;

ALTER SYSTEM

=> ALTER SYSTEM SET log_statement = 'all';

ALTER SYSTEM

=> SELECT pg_reload_conf();

pg_reload_conf
-----
t
(1 row)
```

Начнем новое соединение и локальную транзакцию Read Committed.

```
=> \q

student$ psql fdw_overview

=> BEGIN;

BEGIN

Обращаемся к сторонней таблице:

=> SELECT * FROM test2;
```

```
s
-----
foo
bar
(2 rows)
```

В журнале второго сервера видим:

- установлено новое соединение;
- в нем выставлены некоторые параметры;
- начата транзакция с уровнем изоляции Repeatable Read;
- с помощью курсора прочитаны записи из таблицы test.

```
student$ sudo tail -n 12 /var/log/postgresql/postgresql-16-replica.log
```

```
2025-06-24 05:51:10.841 MSK [106324] [unknown]@[unknown] LOG: connection received:
```

```
host=127.0.0.1 port=49408
2025-06-24 05:51:10.848 MSK [106324] postgres@postgres LOG:  connection authorized:
user=postgres database=postgres application_name=postgres_fdw SSL enabled
(protocol=TLSSv1.3, cipher=TLS_AES_256_GCM_SHA384, bits=256)
2025-06-24 05:51:10.849 MSK [106324] postgres@postgres LOG:  statement: SET search_path =
pg_catalog
2025-06-24 05:51:10.849 MSK [106324] postgres@postgres LOG:  statement: SET timezone =
'GMT'
2025-06-24 05:51:10.849 MSK [106324] postgres@postgres LOG:  statement: SET datestyle =
ISO
2025-06-24 05:51:10.849 MSK [106324] postgres@postgres LOG:  statement: SET intervalstyle
= postgres
2025-06-24 05:51:10.849 MSK [106324] postgres@postgres LOG:  statement: SET
extra_float_digits = 3
2025-06-24 05:51:10.849 MSK [106324] postgres@postgres LOG:  statement: START TRANSACTION
ISOLATION LEVEL REPEATABLE READ
2025-06-24 05:51:10.849 MSK [106324] postgres@postgres LOG:  execute <unnamed>: DECLARE
c1 CURSOR FOR
        SELECT s FROM public.test
2025-06-24 05:51:10.850 MSK [106324] postgres@postgres LOG:  statement: FETCH 100 FROM c1
2025-06-24 05:51:10.850 MSK [106324] postgres@postgres LOG:  statement: CLOSE c1
```

Завершаем локальную транзакцию.

```
=> COMMIT;
```

```
COMMIT
```

В журнале видим, что удаленная транзакция также завершена:

```
student$ sudo tail -n 1 /var/log/postgresql/postgresql-16-replica.log
```

```
2025-06-24 05:51:11.031 MSK [106324] postgres@postgres LOG:  statement: COMMIT TRANSACTION
```

Завершаем сеанс.

```
=> \q
```

В этот момент завершается и соединение с удаленным узлом:

```
student$ sudo tail -n 2 /var/log/postgresql/postgresql-16-replica.log
```

```
2025-06-24 05:51:11.114 MSK [106324] postgres@postgres LOG:  could not receive data from
client: Connection reset by peer
2025-06-24 05:51:11.114 MSK [106324] postgres@postgres LOG:  disconnection: session time:
0:00:00.272 user=postgres database=postgres host=127.0.0.1 port=49408
```

Журнал сообщений можно использовать, чтобы разобраться, как работа внешних средств выглядит с точки зрения СУБД.