

dblink vs fdw jsonb

ChatGPT

postgresql выполнение удаленной функции с передачей и возвращением параметров в формате jsonb что использовать dblink или fd. недостатки. проблемы

В PostgreSQL для выполнения удалённой функции с передачей и возвратом параметров в формате **JSONB** можно использовать как **dblink**, так и **postgres_fdw**. Ниже — сравнение двух подходов, их достоинства, недостатки и потенциальные проблемы.

1. dblink

dblink позволяет выполнять произвольные SQL-запросы на удалённой базе, включая вызов функций.

✓ Преимущества:

- Позволяет **вызвать любую функцию** на удалённой стороне (**SELECT * FROM dblink(...)**).

- Гибкость в передаче параметров, в т.ч. **JSONB**.
- Поддерживает возвращаемые значения любых типов.

✗ Недостатки:

- Требуется явное описание типов (**RETURNING** требует **AS (field type, ...)**).
- Нужно вручную управлять подключениями: строки подключения хранятся в коде или в **pg_service.conf**.
- Отсутствие контроля доступа на уровне SQL-объектов (т.к. работает вне схемы и ролей).
- Не поддерживает **pushdown** оптимизацию.
- Сложнее в отладке и трассировке.
- Сложнее обрабатывать ошибки — всё завернуто в **TEXT**.

💣 Проблемы:

- Безопасность: строки подключения могут содержать пароли.
- Нестабильная сериализация/десериализация сложных структур.
- Ошибки на удалённой стороне могут быть сложно интерпретируемы на локальной.

🌐 2. postgres_fdw

postgres_fdw позволяет подключить удалённую таблицу как локальную и использовать SQL-запросы.

✓ Преимущества:

- Прозрачный доступ к удалённым таблицам.
- Можно вставлять **JSONB** в удалённую таблицу и читать из неё.
- Поддерживает **pushdown запросов**, что повышает производительность.
- Естественная интеграция в структуру схем и ролей PostgreSQL.
- Лучше масштабируется и безопаснее (авторизация через роли, **user mappings**).

✗ Недостатки:

- Нельзя напрямую вызвать функцию (например, `SELECT удалённая_функция()` не работает).
- Придётся использовать **временные таблицы** или представления для обмена данными.
- Могут быть проблемы с типами, если на стороне FDW не поддерживаются кастомные типы (например, расширения).

Проблемы:

- Если функция создаёт побочные эффекты (insert/update/delete), то придётся реализовать обходные пути через `INSERT INTO удалённая_таблица` → функция триггером или правилом.
- **FDW** не поддерживает DML на **представлениях с вызовом функций** — нужно явное обновление таблиц.

✓ Что выбрать?

Требование	Рекомендация
Нужно напрямую вызывать функции	<code>dblink</code>
Вызываются только SQL-операции (insert/select)	<code>postgres_fdw</code>
Требуется безопасность, масштабируемость, контроль доступа	<code>postgres_fdw</code>
Обмен через <code>JSONB</code> , но можно использовать промежуточные таблицы	<code>postgres_fdw</code> лучше
Нужна прозрачная работа с объектами, схемами, ролями	<code>postgres_fdw</code>
Нужно работать с произвольными функциями без обёрток	<code>dblink</code>

Возможный компромисс:

Если нужна функция + fdw:

1. Создайте на удалённой стороне таблицу `tasks_in(jsonb)`, `tasks_out(jsonb)`.
2. На локальной стороне через `INSERT INTO remote.tasks_in(jsonb)` передаёте данные.
3. Удалённая функция триггером читает `tasks_in`, обрабатывает, пишет в `tasks_out`.

4. Локальная сторона читает `SELECT * FROM remote.tasks_out`.

Если хочешь, могу показать пример реализации с `postgres_fdw` — через обмен `JSONB` задачами через таблицы.

поддерживается ли разработчиком `dblink postgresql`

Да, `dblink` официально поддерживается разработчиками PostgreSQL, но:

Статус `dblink`:

- Это **расширение**, входящее в состав официального дистрибутива PostgreSQL.
 - Находится в пакете `contrib` → `postgresql-contrib`.
 - Поддерживается и развивается командой PostgreSQL, но **как расширение, а не как часть ядра**.
-

Особенности поддержки:

- Не приоритетно по сравнению с `postgres_fdw`, который считается **более современным решением**.
 - Обновляется, но **очень редко**, только при необходимости (например, совместимость с новыми версиями).
 - Документация официальная, но минимальная: → <https://www.postgresql.org/docs/current/dblink.html>
-

Что важно знать:

Факт	Подробность
Поддержка	Да, официально входит в PostgreSQL.

Факт	Подробность
Активная разработка	Нет, почти не развивается.
Совместимость	Совместим с текущими версиями PostgreSQL (до 16 включительно).
Альтернатива	<code>postgres_fdw</code> рекомендован как основное средство для межсерверного взаимодействия.

✓ Вывод:

`dblink` поддерживается, можно безопасно использовать, **но если начинаете новый проект или есть возможность — лучше использовать `postgres_fdw`**. Он безопаснее, производительнее и имеет будущее.

`postgres_fdw` как вызов функции на удалённом сервере или хранимую процедуру

В PostgreSQL через `postgres_fdw` **нельзя напрямую вызывать функцию или хранимую процедуру** на удалённом сервере — `SELECT remote_schema.function_name(...)` не будет работать, потому что `postgres_fdw` обменивается **только данными из таблиц**, а не функциями.

Но есть **обходные способы**, и они вполне рабочие:

✓ Способы вызова функции через `postgres_fdw`

🔄 1. Обмен через таблицу + триггер

Сценарий:

- Вы вставляете данные (например, `JSONB`) в удалённую таблицу.
- На удалённой стороне стоит триггер, который вызывает нужную функцию.

- Результат сохраняется в другой таблице, откуда вы его читаете.

Пример:

Удалённый сервер:

sql

```
CREATE TABLE request_queue (  
    id serial PRIMARY KEY,  
    payload jsonb,  
    response jsonb,  
    status text DEFAULT 'new'  
);  
  
-- Функция для обработки  
CREATE OR REPLACE FUNCTION process_request(jsonb)  
RETURNS jsonb AS $$  
BEGIN  
    -- какая-то логика  
    RETURN jsonb_build_object('result', 'ok');  
END;  
$$ LANGUAGE plpgsql;  
  
-- Триггер для автоматической обработки  
CREATE OR REPLACE FUNCTION trg_process()  
RETURNS trigger AS $$  
BEGIN  
    NEW.response := process_request(NEW.payload);  
    NEW.status := 'done';  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER run_processing  
BEFORE INSERT ON request_queue  
FOR EACH ROW EXECUTE FUNCTION trg_process();
```

Локальный сервер:

1. Настроен `postgres_fdw`, подключена таблица `request_queue`.
2. Вставляем данные и читаем результат:

sql

```
-- вставляем запрос
INSERT INTO remote_schema.request_queue (payload)
VALUES ('{"action": "do_something"}'::jsonb)
RETURNING id;

-- потом читаем результат
SELECT response FROM remote_schema.request_queue
WHERE id = <тот id>;
```

2. Создание обёртки-функции на удалённой стороне как SECURITY DEFINER

Можно создать удалённую **таблицу**, в которой **INSERT** вызывает функцию через **BEFORE INSERT** триггер. Это позволяет "обернуть" вызов функции и вызывать её как будто через таблицу.

3. Использовать dblink внутри удалённой функции

Если всё же очень нужно, чтобы локальная функция вызывала функцию на другом сервере, то вы можете **оставить fdw**, но сделать **обёртку внутри удалённой базы** через **dblink** (внутри функции, если нужно обращаться к третьему серверу).

Почему fdw не поддерживает прямой вызов функций?

Потому что **postgres_fdw** реализует доступ к **таблицам**, а не полную виртуализацию SQL API. Он оптимизирован для pushdown'a обычных **SELECT**, **INSERT**, **UPDATE**, **DELETE**, а не вызова процедур.

Вывод:

Цель	Решение
Вызов функции через fdw	Вставка в таблицу с триггером
Хочется SELECT remote_func()	Используйте dblink

Цель	Решение
Массовый вызов или интеграция	Таблица вход + таблица выход
Нужен возврат значения	RETURNING с response -полем

Если хочешь — могу сделать пример со скриптами создания всех объектов (на обеих сторонах), чтобы это сразу работало как "удалённый вызов процедуры через fdw".

вызвать удаленную функцию dblink

Чтобы **вызвать удалённую функцию через dblink**, используется обычный SQL-запрос в виде строки — ты просто вызываешь **SELECT dblink(...)**, передавая SQL, который должен выполняться на **удалённой стороне**.

✓ Синтаксис вызова функции через **dblink**

sql

```
SELECT * FROM dblink(
    'host=... dbname=... user=... password=...',
    'SELECT your_function_name($1::jsonb)'
) AS t(result jsonb);
```

Или если функция возвращает простой тип:

sql

```
SELECT * FROM dblink(  
    'host=localhost dbname=mydb user=myuser password=mypass',  
    'SELECT my_function(''{\"key\": \"value\"}''::jsonb)'  
) AS t(result jsonb);
```

Обрати внимание: **вложенные кавычки** важны (`' '` вместо `'` внутри SQL-строки).

Пример

Допустим, на удалённом сервере есть:

sql

```
CREATE FUNCTION remote_process(data jsonb)  
RETURNS jsonb AS $$  
BEGIN  
    RETURN jsonb_build_object('status', 'ok', 'input', data);  
END;  
$$ LANGUAGE plpgsql;
```

На локальной стороне:

sql

```
SELECT * FROM dblink(  
    'host=192.168.0.10 dbname=remotedb user=remote_user password=secret',  
    'SELECT remote_process(''{\"test\":123}''::jsonb)'  
) AS t(response jsonb);
```

Подводные камни:

Проблема

Решение

Нельзя напрямую передать параметры

Вставляй значения прямо в SQL-строку (с `quote_literal()` или ручным экранированием)

Нужно объявить структуру `AS t(...)`

Всегда указывай ожидаемые поля и типы

Ошибки функции — возвращаются как ошибки dblink

Оборачивай вызов через `DO $$ BEGIN ... EXCEPTION ... END; $$` внутри SQL

Небезопасно хранить пароль

Используй `.pgpass` или `pg_service.conf`

Советы:

1. Лучше обернуть вызов в свою функцию:

sql

```
CREATE FUNCTION call_remote(jsonb) RETURNS jsonb AS $$
DECLARE
    result jsonb;
BEGIN
    SELECT response INTO result FROM dblink(
        'host=... dbname=... user=... password=...',
        'SELECT remote_process('' || data::text || '::jsonb)'
    ) AS t(response jsonb);
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

2. Или использовать **dblink_connect** один раз, а потом работать через **dblink_exec**, **dblink_exec_params**, **dblink_get_result**.

Если хочешь, могу сделать готовый шаблон с параметризацией и безопасным подключением.

с помощью dblink мне нужно с помощью удаленной функции запустить выполнение pgbadger как это делать

Запустить **pgBadger** на удалённой стороне через функцию PostgreSQL и вызвать её через **dblink** — **возможно**, но требует обходных путей, так как:

1. **PostgreSQL** сам не может запускать внешние процессы напрямую, если только не использовать **plpythonu**, **plperl**, **plsh**, **plv8** и т.п.

2. Сам **pgBadger** — это внешняя утилита, которую нужно запускать в **shell**, а не SQL.
-

✓ Возможные варианты

🔧 Вариант 1: Использовать **plpythonu** или **plsh** на удалённой стороне

На удалённой стороне создать **функцию**, которая через **plpythonu** или **plsh** запускает **pgbadger**.

📦 Удалённая сторона (сервер PostgreSQL с установленным **pgbadger**):

1. Убедиться, что есть **plpythonu**:

```
sql
```

```
CREATE EXTENSION IF NOT EXISTS plpythonu;
```

2. Создать функцию, которая запускает **pgBadger**:

```
sql
```

```
CREATE OR REPLACE FUNCTION run_pgbadger(logfile text, outfile text)
RETURNS text AS $$
import subprocess
try:
    cmd = f'pgbadger {logfile} -o {outfile}'
    subprocess.run(cmd, shell=True, check=True)
    return 'pgBadger run OK'
except Exception as e:
    return 'Error: ' + str(e)
$$ LANGUAGE plpythonu;
```

⚠ Это работает **только если суперпользователь** разрешил `p1pythonu`, и только если у PostgreSQL есть доступ к логам.

3. Вызвать её через `dblink`:

sql

```
SELECT * FROM dblink(  
    'host=remotehost dbname=yourdb user=postgres password=secret',  
    'SELECT run_pgbadger(''/var/log/postgresql/postgresql.log'',  
    ''/tmp/report.html'')'  
) AS t(result text);
```

Вариант 2: Через `cron` или фоновую очередь

Если запускать `pgBadger` из SQL слишком опасно или заблокировано:

1. На удалённой стороне сделать таблицу `pgbadger_queue`:

sql

```
CREATE TABLE pgbadger_queue (  
    id serial,  
    logfile text,  
    outfile text,  
    status text default 'new',  
    requested_at timestamp default now()  
);
```

2. Локально вставляете задание через `dblink`:

sql

```
SELECT * FROM dblink(  
    'host=remotehost dbname=yourdb user=postgres password=secret',  
    'INSERT INTO pgbadger_queue (logfile, outfile) VALUES  
(''/var/log/postgresql/postgresql.log'', ''/tmp/report.html'') RETURNING id'  
) AS t(id int);
```

3. На удалённой стороне работает **cron** или демон, который:

- проверяет таблицу,
- запускает **pgbadger**,
- обновляет статус.

Безопасность и ограничения:

Риск	Комментарий
plpythonu небезопасен	Полный доступ к ОС, нужен суперпользователь.
plsh ещё хуже	Позволяет исполнять shell-скрипты.
pgBadger должен быть в \$PATH	Или указывать полный путь.
Права PostgreSQL должны разрешать доступ к логам	PostgreSQL часто запускается от postgres , и лог должен быть читаем.

Рекомендованный путь:

На практике чаще всего:

- Ставят **pgbadger** на сервер,
 - Делают **очередь** в БД (**pgbadger_queue**),
 - Скрипт или демон (на Python/bash) раз в N минут проверяет очередь и запускает **pgbadger**.
-

Хочешь — могу сгенерировать пример такого мини-сервиса (SQL + bash + dblink-вставка).

Хочешь — могу сгенерировать пример такого мини-сервиса (SQL + bash + dblink-вставка).

Отлично! Тогда вот **полноценный минимальный пример** реализации запуска **pgBadger** через очередь в БД + **dblink**. Работает надёжно и безопасно, без **plpythonu**.

ЧАСТЬ 1: Удалённая база PostgreSQL

Шаг 1: Таблица очереди

sql

```
CREATE TABLE pgbadger_queue (  
    id serial PRIMARY KEY,  
    logfile text NOT NULL,  
    outfile text NOT NULL,  
    status text DEFAULT 'new',  
    requested_at timestamp DEFAULT now(),  
    processed_at timestamp  
);
```

ЧАСТЬ 2: Локальный сервер

Шаг 2: Вставка задания через **dblink**

sql

```
SELECT * FROM dblink(  
    'host=192.168.0.10 dbname=remote_db user=remote_user password=secret',  
    $$  
    INSERT INTO pgbadger_queue (logfile, outfile)  
    VALUES ('/var/log/postgresql/postgresql.log',  
    '/tmp/report_2025_04_06.html')  
    RETURNING id  
    $$  
) AS t(id int);
```

ЧАСТЬ 3: Bash-скрипт (демон или cron)

 **run_pgbadger_queue.sh**

bash

```
#!/bin/bash

# Настройки подключения к БД
DBNAME="remote_db"
DBUSER="postgres"
LOGFILE="/var/log/postgresql/pgbadger_runner.log"

# Обрабатываем только новые задания
psql -U $DBUSER -d $DBNAME -At -F $'\t' -c "
SELECT id, logfile, outfile
FROM pgbadger_queue
WHERE status = 'new'
ORDER BY id
LIMIT 1;
" | while IFS=$'\t' read -r id logfile outfile; do

    echo "[$(date)] Running pgBadger for job $id" >> "$LOGFILE"

    # Запуск pgbadger
    if pgbadger "$logfile" -o "$outfile"; then
        # Успешно
        psql -U $DBUSER -d $DBNAME -c "
            UPDATE pgbadger_queue
            SET status = 'done', processed_at = now()
            WHERE id = $id;
        "
        echo "[$(date)] Job $id completed" >> "$LOGFILE"
    else
        # Ошибка
        psql -U $DBUSER -d $DBNAME -c "
            UPDATE pgbadger_queue
            SET status = 'error', processed_at = now()
            WHERE id = $id;
        "
        echo "[$(date)] Job $id failed" >> "$LOGFILE"
    fi
done
```

ЧАСТЬ 4: Планировщик (cron)

Добавь в **crontab** на удалённом сервере:

cron

```
*/5 * * * * /path/to/run_pgbadger_queue.sh
```

✓ Что ты получаешь:

- **dblink** безопасно вставляет задания.
- Логи не нужно передавать — они уже на сервере.
- **pgBadger** запускается в фоновом режиме.
- Отчёты **html** формируются на удалённой стороне.
- Всё логируется, можно контролировать статус.

Если хочешь — могу адаптировать под **systemd** (вместо cron), под Docker, или добавить оповещения.

я выполняю от имени Ф функцию на сервере С в pgAdmin. Почему ошибка при выполнении той же функции С от имени Ф в dblink






Отличный вопрос! Ты столкнулся с **одним из самых частых подводных камней при работе с dblink** — различием в **контексте пользователя и сеанса** между pgAdmin и **dblink**.

📌 Суть проблемы

Когда ты:

- выполняешь функцию на сервере **C** из **pgAdmin** → ты работаешь как пользователь **Ф** напрямую, и все его права, настройки окружения, **search_path**, роли работают как ожидается.
- выполняешь ту же функцию через **dblink** → ты создаёшь новое подключение, которое может иметь:
 - другого пользователя (в строке подключения);
 - другой **search_path**;
 - отсутствующие роли, схемы, расширения;
 - ограничения на доверенность функций (например, **SECURITY DEFINER** не работает как ожидалось).

Частые причины ошибки при вызове функции через **dblink**

Причина	Объяснение	Что делать
 Другой пользователь	В dblink() указываешь user=... — это может быть не Ф	Проверь строку подключения, используй того же пользователя
 Неверный search_path	Внутри dblink -сессии может не быть нужной схемы по умолчанию	Явно укажи схему: SELECT schema.func(...)
 Недостаточно прав	У пользователя, заданного в dblink , нет прав на схему, таблицу, функцию	Временное решение: GRANT EXECUTE ON FUNCTION ... TO ...
 SECURITY DEFINER не срабатывает	SECURITY DEFINER может не применяться так, как ожидалось, особенно если вызывается из dblink внутри SECURITY INVOKER функции	Убедись, что SECURITY DEFINER стоит на нужной функции, или вызывай её напрямую
 pg_hba.conf не даёт права пользователю	Иногда pg_hba.conf запрещает подключение определённым пользователям, даже если права в БД есть	Убедись, что пользователь разрешён на сервере C

Как проверить и исправить

1. Проверь, от какого пользователя идёт вызов через **dblink**

sql

```
SELECT * FROM dblink(  
    'host=... dbname=... user=Φ password=... ',  
    'SELECT current_user, session_user, current_database()'  
) AS t(current_user text, session_user text, db text);
```

→ Убедись, что это тот же пользователь Φ, как в pgAdmin.

2. Проверь **search_path** на стороне dblink:

sql

```
SELECT * FROM dblink(  
    'host=... dbname=... user=Φ password=... ',  
    'SHOW search_path'  
) AS t(path text);
```

→ Если путь не содержит нужной схемы, добавь её в SQL:

sql

```
SET search_path TO myschema;
```

Либо вызывай явно: **SELECT myschema.my_func()**.

3. Временно добавь логирование ошибок в функцию

Если ты вызываешь функцию, и она падает, оберни её так:

```
sql
```

```
BEGIN;  
SELECT dblink(  
    'host=... dbname=... user=Φ password=...',  
    'DO $$ BEGIN PERFORM my_schema.my_func(); EXCEPTION WHEN OTHERS THEN RAISE  
NOTICE ''ERR: %'', SQLERRM; END; $$'  
);  
COMMIT;
```

Это покажет реальную ошибку, которую PostgreSQL может прятать.

✓ Вывод

Ошибка при вызове через **dblink**, а не через pgAdmin:

- Почти всегда из-за **другого пользователя или окружения**.
- Проверь **user**, **search_path**, права и контекст.
- Если нужно, могу помочь разобрать конкретную ошибку (**ERROR: ...**) — пришли её.

Хочешь — можем вместе отладить конкретный случай: покажи код функции и ошибку.

Убедись, что SECURITY DEFINER стоит на нужной функции, или
вызывай её напрямую КАК

Хорошо, давай разберёмся по шагам — как именно **обеспечить вызов функции с нужными правами через SECURITY DEFINER**, особенно при использовании **dblink**.

Цель

Ты хочешь, чтобы функция выполнялась **с правами определённого пользователя**, независимо от того, кто её вызывает через **dblink**.

Что делает SECURITY DEFINER

Если функция помечена как **SECURITY DEFINER**, то она **выполняется с правами того пользователя, который её создал** (а не того, кто её вызывает).

Пример: как правильно настроить SECURITY DEFINER

1. Создай функцию от имени пользователя с нужными правами (например, Φ):

sql

```
CREATE OR REPLACE FUNCTION run_pgbadger_job(logfile text, outfile text)
RETURNS text
LANGUAGE plpgsql
SECURITY DEFINER
AS $$
BEGIN
    -- вставка в очередь
    INSERT INTO pgbadger_queue(logfile, outfile)
    VALUES (logfile, outfile);

    RETURN 'Задание принято';
END;
$;
```

2. Разреши доступ другим пользователям на вызов этой функции:

sql

```
GRANT EXECUTE ON FUNCTION run_pgbadger_job(text, text) TO PUBLIC;  
-- или конкретному пользователю, если хочешь ограничить  
-- GRANT EXECUTE ON FUNCTION run_pgbadger_job(text, text) TO dblink_user;
```

3. Убедись, что `search_path` не испортит выполнение

!! ВАЖНО: `SECURITY DEFINER` функции уязвимы, если `search_path` включает `public`, потому что злоумышленник может подменить объекты!

Поэтому **установи `search_path` прямо внутри функции:**

sql

```
SET search_path TO myschema, pg_temp;
```



Обновлённая функция:

sql

```
CREATE OR REPLACE FUNCTION run_pgbadger_job(logfile text, outfile text)
RETURNS text
LANGUAGE plpgsql
SECURITY DEFINER
AS $$
BEGIN
    PERFORM set_config('search_path', 'myschema, pg_temp', false);

    INSERT INTO pgbadger_queue(logfile, outfile)
    VALUES (logfile, outfile);

    RETURN 'Задание принято';
END;
$$;
```

Вызов через dblink

Теперь ты можешь вызвать её от **любого пользователя**, даже у которого нет прав на `pgbadger_queue` напрямую:

sql

```
SELECT * FROM dblink(
    'host=... dbname=... user=обычный_пользователь password=...',
    'SELECT run_pgbadger_job('/var/log/postgresql/postgresql.log',
    '/tmp/report.html')'
) AS t(result text);
```

✓ Работает, потому что:

- функция `run_pgbadger_job` помечена как `SECURITY DEFINER`;
- вызывается с правами **создателя функции**, а не dblink-пользователя;

- `search_path` жёстко зафиксирован внутри.

Резюме: что надо, чтобы **SECURITY DEFINER** сработал через **dblink**

Шаг Что сделать

- | | |
|---|-------------------------------------------------------------------|
| 1 | Создать функцию с SECURITY DEFINER от нужного пользователя |
| 2 | Внутри функции явно установить <code>search_path</code> |
| 3 | Выдать GRANT EXECUTE нужным пользователям |
| 4 | Вызывать её через <code>dblink</code> , не волнуясь о правах |
-

Если хочешь, я могу помочь сгенерировать весь SQL скрипт под твою структуру — просто скажи, как зовётся функция и какие параметры она принимает.

```
DO $$ DECLARE id_barn_flag INTEGER; time_start BIGINT; omega
RECORD; json_flight JSONB; err_mess TEXT; err_det TEXT; err_cd TEXT;
log_synthesized_name TEXT; ts_current TIMESTAMP =
CURRENT_TIMESTAMP; start_time TIMESTAMP; end_time TIMESTAMP;
time_diff INTERVAL; result_json JSONB; log_size INTEGER; log_slice_size
INTEGER; BEGIN IF NOW()::TIME BETWEEN TIME '00:00:00' AND TIME
'00:10:00' THEN log_synthesized_name = FORMAT('postgresql-%s.log',
TO_CHAR((CURRENT_DATE - INTERVAL '1 day'), 'Dy')); start_time =
TO_TIMESTAMP( TO_CHAR((CURRENT_DATE - INTERVAL '1
day')::TIMESTAMP + TIME '22:00:00', 'YYYY-MM-DD HH24:MI:SS'), 'YYYY-
MM-DD HH24:MI:SS'); end_time = TO_TIMESTAMP(
TO_CHAR((CURRENT_DATE - INTERVAL '1 day')::TIMESTAMP + TIME
'23:59:59', 'YYYY-MM-DD HH24:MI:SS'), 'YYYY-MM-DD HH24:MI:SS');
ELSE log_synthesized_name = FORMAT('postgresql-%s.log',
TO_CHAR(CURRENT_DATE, 'Dy')); END IF; ts_current = (ts_current AT
TIME ZONE 'UTC')::TIMESTAMP; time_diff = ts_current -
DATE_TRUNC('hour', ts_current); IF (time_diff BETWEEN INTERVAL '-5
minutes' AND INTERVAL '5 minutes') THEN start_time =
DATE_TRUNC('hour', ts_current) - INTERVAL '2 hours'; end_time =
DATE_TRUNC('hour', ts_current); RAISE NOTICE 'Interval: % - %',
start_time, end_time; ELSE NULL; END IF; IF start_time IS NULL THEN
```

```

start_time = DATE_TRUNC('hour', ts_current) - INTERVAL '2 hours';
end_time = DATE_TRUNC('hour', ts_current); END IF; CREATE TEMP
TABLE tmp_info_table ( conn_name TEXT, serv_id INTEGER, serv_port
INTEGER, serv_host TEXT, compl_name TEXT, out_dir TEXT, out_file
TEXT, pgbg_path TEXT, log_synthesized_name TEXT, start_time
TIMESTAMP, end_time TIMESTAMP ) ON COMMIT DROP; INSERT INTO
tmp_info_table SELECT 'y_connect' AS conn_name, s.Pk_Id_Conn AS
serv_id, s.Conn_Port AS serv_port, s.Conn_Host AS serv_host,
REGEXP_REPLACE( FORMAT('%s--%s', s.Conn_Host, s.Conn_Port),
'\.maxbit\.private', '', 'g' ) || '--' || TO_CHAR(NOW() AT TIME ZONE 'UTC',
'YYYY-MM-DD--HH24-MI') || '.html', pi.Path_Out_Dir AS out_dir,
pi.Path_Out_Log_File AS out_file, pi.Path_Pgbg AS pgbg_path,
log_synthesized_name, start_time AT TIME ZONE 'UTC', end_time AT
TIME ZONE 'UTC' FROM Robohub.Reference."Servers" s JOIN Pg_Ini pi
ON pi.Fk_Pk_Id_Conn = s.Pk_Id_Conn WHERE (s.Switch_Serv &
B'00100000') = B'00100000'; FOR omega IN SELECT * FROM
tmp_info_table LOOP BEGIN IF omega.conn_name IN (SELECT
UNNEST(Robohub.Public.Dblink_Get_Connections())) THEN PERFORM
Robohub.Public.Dblink_Disconnect(omega.conn_name); END IF; INSERT
INTO Pgbadger_Repo_Slicer.Pg_Barn (Fk_Pk_Id_Conn, Log_Name)
VALUES (omega.serv_id, omega.compl_name) RETURNING Pk_Id_Barn
INTO id_barn_flag; json_flight = JSONB_BUILD_OBJECT( 'pgbg_path',
omega.pgbg_path, 'log_file', omega.out_file || log_synthesized_name,
'out_dir_slice', omega.out_dir, 'out_file_slice', omega.compl_name,
'begin_dattime', TO_CHAR(omega.start_time, 'YYYY-MM-DD
HH24:MI:SS'), 'end_dattime', TO_CHAR(omega.end_time, 'YYYY-MM-DD
HH24:MI:SS') ); time_start = EXTRACT(EPOCH FROM
CLOCK_TIMESTAMP()); PERFORM Robohub.Public.Dblink_Connect(
omega.conn_name, FORMAT( 'dbname=%s user=%s password=%s
host=%s port=%s', 'postgres', 'robo_sudo', '%dFgH8!zX4&kLmT2', --
'postgres', 'gtimofeyev', 'd56f2c4eFDnofa1w', omega.serv_host,
omega.serv_port::INT ) ); result_json =
Pgbadger_Report_Slicer(json_flight); RAISE NOTICE 'Результат: %',
result_json ->> 'pgbadger_command'; log_size = ( SELECT (elem ->>
'size')::INTEGER FROM JSONB_ARRAY_ELEMENTS(result_json -> 'logs')
AS elem WHERE elem ->> 'name' = json_flight ->> 'log_file' LIMIT 1 );
log_slice_size = COALESCE((result_json ->> 'file_size')::INTEGER, 0);
UPDATE Pgbadger_Repo_Slicer.Pg_Barn SET Log_Timer_Slicing =
EXTRACT(EPOCH FROM CLOCK_TIMESTAMP()) - time_start, Log_Size =
COALESCE(log_size, 0), Log_Slice_Size = log_slice_size,
Log_Slice_Name = omega.compl_name -- Указываем имя
сгенерированного отчета WHERE Pk_Id_Barn = id_barn_flag; IF
omega.conn_name IN (SELECT
UNNEST(Robohub.Public.Dblink_Get_Connections())) THEN PERFORM
Robohub.Public.Dblink_Disconnect(omega.conn_name); END IF;

```

```

EXCEPTION WHEN OTHERS THEN GET STACKED DIAGNOSTICS
err_mess = MESSAGE_TEXT, err_det = PG_EXCEPTION_DETAIL, err_cd
= RETURNED_SQLSTATE; INSERT INTO
Robohub.Pgbadger_Repo_Slicer."Errors" (Fk_Pk_Id_Conn,
Slice_Err_Code, Slice_Err_Detail, Slice_Err_Mess, Slice_Now_Ins)
VALUES (omega.serv_id, err_cd, err_det, err_mess, NOW()); RETURN;
END; END LOOP; END; $$; CREATE OR REPLACE FUNCTION
Pgbadger_Report_Slicer(input_json JSONB) RETURNS JSONB AS $$
DECLARE output_json JSONB; str_complete TEXT; cmd TEXT; file_size
BIGINT; BEGIN SELECT JSONB_AGG( JSONB_BUILD_OBJECT( 'name',
'name', 'size', 'size', 'modification', 'modification' ) ) INTO output_json
FROM Pg_Ls_Logdir() WHERE "modification"::DATE = CURRENT_DATE
AND RIGHT("name", 4) = '.log'; str_complete := FORMAT( 'pgbadger --
begin "%s" --end "%s" --outdir %s --outfile %s %s', input_json ->>
'begin_dattime', input_json ->> 'end_dattime', input_json ->> 'out_dir_slice',
input_json ->> 'out_file_slice', input_json ->> 'log_file' ); cmd :=
FORMAT('COPY (SELECT "NULL") TO PROGRAM %L', str_complete);
BEGIN EXECUTE cmd; file_size := (pg_stat_file(input_json ->>
'out_dir_slice' || '/' || input_json ->> 'out_file_slice')).size::BIGINT; RAISE
NOTICE 'Команда успешно выполнена: %', str_complete; RAISE NOTICE
'Размер файла: %', file_size; EXCEPTION WHEN OTHERS THEN RAISE
NOTICE 'Ошибка при выполнении команды: %', SQLERRM; file_size :=
-1; -- размер файла в -1 в случае ошибки END; RETURN
JSONB_BUILD_OBJECT( 'logs', COALESCE(output_json, '[]'::JSONB),
'pgbadger_command', str_complete, 'file_size', file_size ); END; $$
LANGUAGE Plpgsql; вот конкретный пример

```

Отлично, ты выдал мощный и продуманный пример — видно, что архитектура хорошо проработана. Теперь — по сути: **почему при вызове через `dblink` может не сработать `Pgbadger_Report_Slicer`, если она работает напрямую от имени того же пользователя?**



Возможные причины, исходя из твоего кода:

1. `Pgbadger_Report_Slicer` вызывается на локальном сервере, а не удалённом

В твоём коде `result_json = Pgbadger_Report_Slicer(json_flight);` вызывается локально, но ты подключаешься к удалённому серверу через `dblink_connect`, и сам `pgbadger` запускается локально!

 **Решение:** если ты хочешь, чтобы `pgbadger` запускался на удалённой стороне, тебе нужно вызвать функцию `Pgbadger_Report_Slicer` через `dblink_exec` или `dblink`.

2. Как правильно вызвать `Pgbadger_Report_Slicer` на удалённой стороне:

Ты можешь заменить строку:


```
sql
```

```
result_json = Pgbadger_Report_Slicer(json_flight);
```

на вызов через `dblink`:

```
sql
```

```
SELECT * INTO result_json FROM dblink(
    omega.conn_name,
    $$SELECT Pgbadger_Report_Slicer($$ || quote_literal(json_flight::text)
    || $$::jsonb)$$
) AS t(result_json jsonb);
```

 Здесь:

- мы сериализуем `json_flight` в строку;
- передаём в `dblink`;
- и приводим результат к типу `jsonb`.

3. Что ещё проверить:

🔒 Функция `Pgbadger_Report_Slicer` должна быть `SECURITY DEFINER`, если она работает с `COPY TO PROGRAM`:

sql

```
CREATE OR REPLACE FUNCTION Pgbadger_Report_Slicer(input_json JSONB)
RETURNS JSONB
LANGUAGE plpgsql
SECURITY DEFINER
AS $$
...
$$;
```

⚠️ Убедись, что пользователь, от имени которого выполняется `dblink`, имеет разрешение на:

- `EXECUTE` этой функции;
 - использование `COPY TO PROGRAM` (это требует `superuser` или специальной настройки `pg_execute_server_program` в PostgreSQL 15+).
-

💡 **Альтернатива: оставить `pgbadger` запускаться локально**

Если тебе нужно именно **собрать логи с удалённого сервера**, но запускать `pgbadger` на **локальном**, то оставь текущую архитектуру, **но проверь**, что `log_file` (т.е. файл журнала) действительно **доступен локально**. Потому что сейчас ты его указываешь как:

sql

```
'log_file', omega.out_file || log_synthesized_name,
```

Если он физически **на удалённой машине**, локальный запуск `pgbadger` его просто не найдёт.

✓ Что бы я предложил

Если ты хочешь, чтобы `pgbadger` запускался на **удалённом сервере**, то:

1. Сделай функцию `Pgbadger_Report_Slicer` доступной на **удалённой стороне**.
2. Вызывай её через `dblink`.
3. Передавай `jsonb` как `text`, и приводи его обратно в `::jsonb`.

Если наоборот — тебе важно запускать анализ **локально**, но с данными с удалённой стороны, то нужно:

- либо монтировать удалённый лог через `sshfs`, `rsync` и т.д.;
 - либо сначала копировать лог файлы локально (по `cron` или другим сервисом), а затем запускать `pgbadger`.
-

Хочешь — могу переписать кусок под вызов `Pgbadger_Report_Slicer` на **удалённой стороне через `dblink`**, со всеми кавычками и безопасным исполнением.

```
произвести подробный анализ CREATE FUNCTION
check_password(uname TEXT, pass TEXT) RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN; BEGIN SELECT (pwd = $2) INTO passed
FROM pwds WHERE username = $1; RETURN passed; END; $$
LANGUAGE plpgsql SECURITY DEFINER -- Установить безопасный путь
поиска: сначала доверенная схема(ы), затем 'pg_temp'. SET
search_path = admin, pg_temp;
```

Конечно! Давай подробно разберём каждую часть функции `check_password`, начиная от её структуры и назначения до вопросов безопасности и потенциальных уязвимостей.

Общая цель функции

sql

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN
```

Эта функция предназначена для **проверки логина и пароля** по таблице **pwds**.
Принимает:

- **uname** — имя пользователя,
- **pass** — предполагаемый пароль.

Возвращает **TRUE** или **FALSE** — в зависимости от того, найден ли пользователь с таким паролем.

Тело функции — разбор по шагам

pl

```
DECLARE passed BOOLEAN;
```

Создаётся переменная **passed**, которая будет хранить результат проверки.

```
plpgsql
```

```
SELECT (pwd = $2) INTO passed  
FROM pwds  
WHERE username = $1;
```

Здесь происходит сравнение:

- `$1 = uname`
- `$2 = pass`

Оператор `(pwd = $2)` вернёт `TRUE`, `FALSE`, или `NULL`. То есть:

- если пользователь найден и пароль совпадает → `TRUE`;
- найден, но пароль не совпадает → `FALSE`;
- не найден → `NULL`.

✓ Хорошо: используется `SELECT ... INTO`, что удобно для хранения результата. ⚠ Однако: если в таблице `pwds` больше одной строки с таким `username`, то будет ошибка `more than one row returned by a subquery used as an expression`.

```
plpgsql
```

```
RETURN passed;
```


Функция возвращает результат — `TRUE`, `FALSE` или `NULL` (если пользователь не найден).

sql

```
LANGUAGE plpgsql  
SECURITY DEFINER
```

SECURITY DEFINER означает, что функция выполняется **от имени пользователя, который её создал** — а не вызывающего.

Это нужно, если вызывающий пользователь **не должен иметь прямой доступ** к таблице **pwds**, но при этом должен иметь возможность проверить пароль.

|  Но это **опасно**, если не настроен правильно **search_path** (см. ниже)!

 **SET search_path = admin, pg_temp — что делает и почему важно**

sql

```
SET search_path = admin, pg_temp;
```

Когда используется **SECURITY DEFINER**, критически важно **зафиксировать search_path**, чтобы избежать подмены объектов (SQL-инъекций через схемы).

Например, если пользователь создаст свою функцию **pwd()** в схеме **public**, а **search_path** будет включать **public** перед **admin**, то при вызове будет использована **вредоносная функция**, а не настоящая таблица/функция из **admin**.

 Здесь фиксируется:

1. **admin** — доверенная схема, где находится **pwds**.

2. `pg_temp` — разрешена временная схема, которая необходима, если функция внутри что-то временно создаёт (хотя здесь это не нужно — можно опустить `pg_temp`).

Безопасность — потенциальные проблемы и как улучшить


Вопрос	Анализ
Пароли в открытом виде?	⚠ Да. Таблица <code>pws</code> хранит пароли в явном виде , сравнение происходит напрямую — это очень опасно .
Решение	Использовать хеши паролей: <code>crypt</code> , <code>pgcrypto</code> , <code>digest()</code> и т.д.
SQL-инъекции	🚫 Нет — передача параметров корректная (<code>\$1</code> , <code>\$2</code>)
Уязвимость через <code>search_path</code>	✅ Замкнута — <code>SET search_path = admin, pg_temp</code>
Множественные строки	⚠ Возможна ошибка, если <code>username</code> не уникален
NULL как результат	🤔 Если пользователь не найден — будет <code>NULL</code> , что может быть нежелательным поведением (лучше всегда возвращать <code>FALSE</code>)

✅ Улучшенный вариант (с учётом безопасности)

sql

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN
LANGUAGE plpgsql
SECURITY DEFINER
SET search_path = admin, pg_temp
AS $$
DECLARE
    passed BOOLEAN := FALSE;
BEGIN
    SELECT (pwd_hash = crypt(pass, pwd_hash))
    INTO passed
    FROM pwds
    WHERE username = uname;

    RETURN COALESCE(passed, FALSE);
END;
$$;
```

 Здесь:

- `pwd_hash` — хеш пароля, заранее сгенерированный (например, с помощью `crypt('secret', gen_salt('bf'))`)
- используется `crypt(pass, pwd_hash)` для проверки
- даже если пользователь не найден — вернётся `FALSE`, а не `NULL`.

Резюме

Плюсы

`SECURITY DEFINER` и `search_path` защищены

Простая и понятная логика

Защита от SQL-инъекций

Легко расширить

Минусы

Пароли хранятся в открытом виде

Нет гарантии уникальности username

Может вернуть `NULL`, что не всегда ожидаемо

Не проверяет активность пользователя и др. атрибуты

Хочешь — могу сгенерировать полную версию безопасной проверки с `pgcrypto`, регистрацией и входом.

Да

Вот улучшенная версия функции с использованием безопасных хешей для паролей с использованием модуля `pgcrypto`. Мы будем использовать хеширование паролей и функцию `crypt` для их сравнения.

Улучшенная версия функции с безопасным хешированием паролей

1. **Пароли** будут храниться в виде хешей, например, с использованием алгоритма Blowfish (который предоставляет `pgcrypto`).
2. Для проверки пароля будет использоваться функция `crypt`, которая сравнивает хешированный пароль с хешем из базы данных.

Действия:

1. Установим зависимость от модуля `pgcrypto`, если он ещё не установлен:

```
sql
```

```
CREATE EXTENSION IF NOT EXISTS pgcrypto;
```

2. Определим новую функцию, которая будет безопасно проверять пароли.
-

Новый SQL-код для функции `check_password`

sql

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN
LANGUAGE plpgsql
SECURITY DEFINER
SET search_path = admin, pg_temp
AS $$
DECLARE
    passed BOOLEAN := FALSE;
BEGIN
    -- Пытаемся найти хеш пароля пользователя в таблице pwds
    SELECT (pwd_hash = crypt(pass, pwd_hash))
    INTO passed
    FROM pwds
    WHERE username = uname;

    -- Возвращаем TRUE, если пароли совпадают, или FALSE, если не совпадают/
    пользователь не найден
    RETURN COALESCE(passed, FALSE);
END;
$$;
```

Разбор изменений:

1. Использование `crypt`:

- Функция `crypt(pass, pwd_hash)` используется для хеширования введённого пароля `pass` и сравнения его с хешом пароля, который хранится в поле `pwd_hash` таблицы `pwds`.
- Хеш пароля, сохранённый в базе данных, был сгенерирован заранее, например, с использованием `crypt('secret', gen_salt('bf'))`, где `'bf'` означает использование алгоритма Blowfish.

2. Использование **COALESCE**:

Если пользователь с таким логином не найден в таблице **pwds**, то переменная **passed** останется **NULL**. Мы используем **COALESCE(passed, FALSE)**, чтобы гарантировать возврат **FALSE** в случае, если пользователя не существует или пароль неверный.

3. Безопасность:

- Хранение паролей как хешей и использование **crypt** для их проверки значительно повышает безопасность.
- Функция работает с параметрами через плейсхолдеры (**\$1**, **\$2**), что защищает от SQL-инъекций.

4. Реализация с **SECURITY DEFINER**:

Мы сохранили использование **SECURITY DEFINER**, что позволяет этой функции быть выполненной с правами владельца функции, а не того пользователя, который её вызывает. Это особенно важно, если функция работает с данными, которые должны быть защищены.

Пример сохранения пароля в базе данных

Для сохранения пароля в виде хеша (например, при регистрации пользователя), мы можем использовать такой SQL-запрос:

```
sql
```

```
INSERT INTO pwds (username, pwd_hash)
VALUES ('user1', crypt('password123', gen_salt('bf')));
```

Здесь:

- **crypt('password123', gen_salt('bf'))** генерирует хеш пароля с использованием алгоритма Blowfish.
- Этот хеш и сохраняется в поле **pwd_hash** таблицы **pwds**.

Дополнительные шаги для улучшения безопасности

1. Защита с использованием соли:

Метод `gen_salt('bf')` генерирует соль для алгоритма Blowfish. Это увеличивает стойкость пароля к атакам с помощью радужных таблиц.

2. Регулярные проверки безопасности:

Рекомендуется регулярно проверять уязвимости PostgreSQL, поддерживать актуальность используемых алгоритмов и солей, а также следить за безопасностью на уровне сервера.

Теперь твоя система для проверки паролей будет более безопасной и устойчивой к атакам. Хеширование паролей с использованием `crypt` делает их невосприимчивыми к утечкам данных.

Если нужно больше деталей или примеров использования — скажи!