Практические работы по дисциплине «Технологии обработки транзакций клиент-серверных приложений» направления подготовки бакалавриата 09.03.04 «Программная инженерия»

Практическая работа №11

Транзакции. Языки программирования

Теория для понимания практики:

Темы

- Языки серверного программирования
- Доверенные и недоверенные языки
- Подключение нового языка
- Трансформации типов
- Интерфейс SPI для работы с базой
- Зачем нужны языки и из чего можно выбирать

Языки программирования

«Встроенные»

- C
- SQL

Стандартные (поддержка сообщества)

- PL/pgSQLPL/Perl, PL/Python, PL/Tcl

Сторонние

- PL/Java, PL/V8, PL/R, PL/Lua, ...

Разумеется, к PostgreSQL можно обращаться «извне» из любых языков программирования (ЯП), для которых реализована поддержка клиент-серверного протокола. Но разные ЯП можно использовать и для программирования на стороне сервера: для написания хранимых функций и процедур, триггеров и т. п.

В PostgreSQL «встроены» два ЯП: C, на котором написана вся система, и SQL. Эти языки доступны всегда.

В систему также входят четыре ЯП, которые имеются в стандартной поставке и поддерживаются сообществом. С одним из них – PL/pgSQL – мы

уже хорошо знакомы. Он доступен по умолчанию и наиболее часто используется на стороне сервера. Другие языки – PL/Perl, PL/Python и PL/Tcl – надо устанавливать как расширения. Такой (несколько странный) выбор языков обусловлен историческими причинами.

В документации языки, отличные от C и SQL, называются процедурными, отсюда и приставка «PL» в названиях. Неудачное название: никто не мешает подключить к PostgreSQL функциональный или еще какой-нибудь язык. Поэтому лучше понимать PL как programming language, а не как procedural language.

PostgreSQL – расширяемая система, поэтому имеется возможность добавления и других ЯП.

Доверенные языки

Доверенные

- гарантируют работу пользователя в рамках выданного доступа
- ограничено взаимодействие с окружением и внутренностями СУБД, язык должен позволять работать в «песочнице»
- по умолчанию доступ для всех пользователей

Недоверенные

- доступна полная функциональность языка
- доступ только у суперпользователей
- обычно к названию языка добавляют «u»: plperl → plperlu

Все ЯП можно разделить на доверенные (trusted) и недоверенные (untrusted).

Доверенные языки должны соблюдать ограничения доступа, установленные в системе. С помощью функций, написанных на таких языках, пользователь не должен суметь получить доступ к чему-либо, если такой доступ ему не предоставлен.

По сути это означает, что язык должен ограничивать взаимодействие пользователя с окружением (например, с операционной системой) ис внутренностями СУБД (чтобы нельзя было обойти обычные проверки доступа). Не все реализации языков умеют работать в таком режиме

«песочницы».

Использование доверенного языка является безопасным, поэтому доступ к нему получают все пользователи (для роли public выдается привилегия usage на язык).

Недоверенные языки не имеют никаких ограничений. В частности, функция на таком языке может выполнять любые операции в ОС с правами пользователя, запустившего сервер базы данных. Это может быть небезопасным, поэтому доступ к такому языку имеют только суперпользователи PostgreSQL.

Напомним, что, если необходимо, суперпользователь может создать функцию на недоверенном языке с указанием SECURITY DEFINER и выдать право на ее исполнение обычным пользователям.

Практика 1. Выполняется на вашей БД по аналогии нижеприведенной.

```
=> CREATE DATABASE ext_languages;
CREATE DATABASE
=> \c ext_languages
You are now connected to database "ext_languages" as user
"student".
```

Проверим список установленных языков:

```
=> \dL
                        List of languages
  Name   |  Owner   | Trusted |         Description
---------+----------+---------+------------------------------
 plpgsql | postgres | t       | PL/pgSQL procedural language
(1 row)
```

По умолчанию установлен только PL/pgSQL (C и SQL не в счет).

Новые языки принято оформлять как расширения. Вот какие доступны для установки:

```
=> SELECT name, comment, installed_version
FROM pg_available_extensions
```

```
WHERE name LIKE 'pl%'
ORDER BY name;
    name    |                  comment                  |
installed_version
------------+-------------------------------------------+-------
 plperl     | PL/Perl procedural language               |
 plperlu    | PL/PerlU untrusted procedural language    |
 plpgsql    | PL/pgSQL procedural language              | 1.0
 plpython3u | PL/Python3U untrusted procedural language |
 plsh       | PL/sh procedural language                 |
 plxslt     | PL/XSLT procedural language               |
(6 rows)
```

Первые четыре — из числа стандартных, а с двумя последними мы познакомимся позже.

Установим в текущую базу данных два варианта языка PL/Perl: plperl (доверенный) и plperlu (недоверенный):

```
=> CREATE EXTENSION plperl;
CREATE EXTENSION
=> CREATE EXTENSION plperlu;
CREATE EXTENSION
=> \dL
                            List of languages
  Name   |  Owner   | Trusted |              Description
---------+----------+---------+-------------------------------
 plperl  | student  | t       | PL/Perl procedural language
 plperlu | student  | f       | PL/PerlU untrusted procedural
language
 plpgsql | postgres | t       | PL/pgSQL procedural language
(3 rows)
```

Чтобы языки автоматически появлялись во всех новых базах данных, расширения нужно установить в БД template1.

Недоверенный язык не имеет ограничений. Например, можно создать

функцию, читающую любой файл (аналогично штатной функции pg_read_file):

```
=> CREATE FUNCTION read_file_untrusted(fname text) RETURNS SETOF
text
AS $perl$
    my ($fname) = @_;
    open FILE, $fname or die "Cannot open file";
    chomp(my @f = <FILE>);
    close FILE;
    return \@f;
$perl$ LANGUAGE plperlu VOLATILE;
CREATE FUNCTION
=> SELECT * FROM read_file_untrusted('/etc/passwd') LIMIT 3;
               read_file_untrusted
-------------------------------------------------
 root:x:0:0:root:/root:/bin/bash
 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
 bin:x:2:2:bin:/bin:/usr/sbin/nologin
(3 rows)
```

Что будет, если попробовать сделать то же самое на доверенном языке?

```
=> CREATE FUNCTION read_file_trusted(fname text) RETURNS SETOF
text
AS $perl$
    my ($fname) = @_;
    open FILE, $fname or die "Cannot open file";
    chomp(my @f = <FILE>);
    close FILE;
    return \@f;
$perl$ LANGUAGE plperl VOLATILE;
ERROR:  'open' trapped by operation mask at line 3.
CONTEXT:  compilation of PL/Perl function "read_file_trusted"
```

Вызов open (в числе прочего) запрещен в доверенном языке.

Подключение нового языка

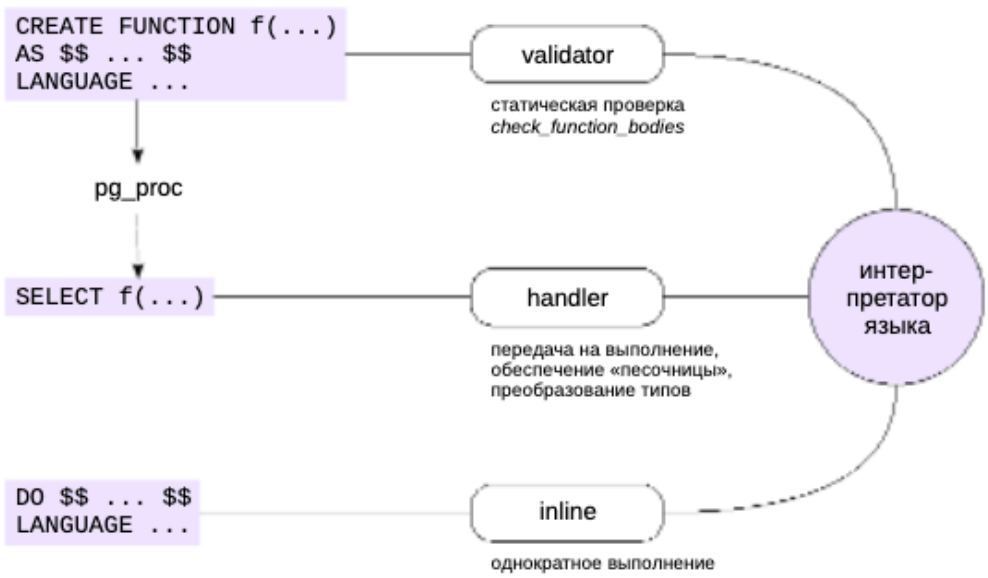

Рисунок 1 – три функции при подключении языка

Интерфейс для подключения нового ЯП для использования на стороне сервера включает всего три функции.

При создании хранимой подпрограммы (функции или процедуры), ее код проверяется на наличие ошибок (validator). Если проверка проходит успешно, подпрограмма сохраняется в системном каталоге. Как мы знаем, текст подпрограммы записывается в виде обычной строки, и ровно эта строка и сохраняется. Таким образом, компиляция не поддерживается (что не исключает использование JIT-компиляции). Проверка может отключаться параметром check_function_bodies; утилита pg_dump пользуется этим, чтобы уменьшить зависимостьот порядка создания объектов.

При вызове подпрограммы ее текст передается интерпретатору на выполнение, а полученный ответ возвращается в систему (handler). Если язык объявлен доверенным, интерпретатор должен запускатьсяв «песочнице». Здесь также решается важная задача преобразования типов. Не все типы SQL могут иметь аналоги в ЯП. Обычный подходв этом случае – передача текстового представления типа, но реализация может также учитывать определенные пользователем трансформации типов.

При выполнении SQL-команды DO интерпретатору передается код для однократного выполнения (inline).

Из этих трех функций только функция-handler является обязательной,

остальные могут отсутствовать.

Практика 2. Выполняется на вашей БД по аналогии нижеприведенной.

Если заглянуть, что выполняет команда CREATE EXTENSION, для недоверенного языка мы увидим примерно следующее:

```
CREATE LANGUAGE plperlu
  HANDLER plperlu_call_handler
  INLINE plperlu_inline_handler
  VALIDATOR plperlu_validator;
```

И для недоверенного (обратите внимание на слово TRUSTED):

```
CREATE TRUSTED LANGUAGE plperl
  HANDLER plperl_call_handler
  INLINE plperl_inline_handler
  VALIDATOR plperl_validator;
```

В этой команде указываются имена функций, реализующих точки входа основного обработчика, обработчика для DO и проверки.

Установим еще один язык — PL/Python. Он доступен только как недоверенный:

```
=> CREATE EXTENSION plpython3u;
CREATE EXTENSION
```

На его примере посмотрим, как происходит преобразование между системой типов SQL и системой типов языка. Для многих типов предусмотрены преобразования:

```
=> CREATE FUNCTION test_py_types(n numeric, b boolean, s text, a
int[])
RETURNS void AS $python$
    plpy.info(n, type(n))
    plpy.info(b, type(b))
    plpy.info(s, type(s))
```

```
    plpy.info(a, type(a))
$python$ LANGUAGE plpython3u IMMUTABLE;
CREATE FUNCTION
=> SELECT test_py_types(42,true,'foo',ARRAY[1,2,3]);
INFO:  (Decimal('42'), <class 'decimal.Decimal'>)
INFO:  (True, <class 'bool'>)
INFO:  ('foo', <class 'str'>)
INFO:  ([1, 2, 3], <class 'list'>)
 test_py_types
---------------

(1 row)
```

А что мы увидим в таком случае?

```
=> CREATE FUNCTION test_py_jsonb(j jsonb)
RETURNS jsonb AS $python$
    plpy.info(j, type(j))
    return j
$python$ LANGUAGE plpython3u IMMUTABLE;
CREATE FUNCTION
=> SELECT test_py_jsonb('{ "foo": "bar" }'::jsonb);
INFO:  ('{"foo": "bar"}', <class 'str'>)
 test_py_jsonb
----------------
 {"foo": "bar"}
(1 row)
```

Здесь SQL-тип json был передан в функцию как строка, а возвращаемое значение было вновь преобразовано в jsonb из текстового представления.

Чтобы помочь обработчику языка, можно создать дополнительные трансформации типов. Для нашего случая есть подходящее расширение:

```
=> CREATE EXTENSION jsonb_plpython3u;
```

```
CREATE EXTENSION
```

Фактически оно создает трансформацию таким образом (что позволяет передавать тип jsonb и в Python, и обратно в SQL):

```
CREATE TRANSFORM FOR jsonb LANGUAGE plpython3u (
    FROM SQL WITH FUNCTION jsonb_to_plpython3(internal),
    TO SQL WITH FUNCTION plpython3_to_jsonb(internal)
);
```

Трансформацию необходимо явно указать в определении функции:

```
=> CREATE OR REPLACE FUNCTION test_py_jsonb(j jsonb)
RETURNS jsonb
TRANSFORM FOR TYPE jsonb -- использовать трансформацию
AS $python$
    plpy.info(j, type(j))
    return j
$python$ LANGUAGE plpython3u IMMUTABLE;
CREATE FUNCTION
=> SELECT test_py_jsonb('{ "foo": "bar" }'::jsonb);
INFO:  ({'foo': 'bar'}, <class 'dict'>)
 test_py_jsonb
---------------
 {"foo": "bar"}
(1 row)
```

Теперь SQL-тип jsonb передается в Python как тип dict — словарь (ассоциативный массив).

Функции обработчика языка и трансформации можно написать только на C, поэтому мы не будем рассматривать, как это делается.

Задачи

Задачами являются:

- Обработка информации, хранимой в базе данных
  - подпрограмма всегда выполняется в контексте подключения к БД

    - PL/pgSQL интегрирован с SQL
    - для остальных — интерфейс серверного программирования (SPI)
- Вычисления, не связанные с базой данных
    - возможности PL/pgSQL сильно ограничены
    - эффективность
    - удобство использования
    - наличие готовых библиотек
    - специализированные задачи

Задачи, для решения которых используются хранимые подпрограммы, можно условно поделить на две группы.

В первую входит работа с информацией, которая содержится внутри базы данных. Здесь любые хранимые подпрограммы выигрывают у внешних (клиентских) программ, поскольку они находятся ближе к данным: во-первых, не требуется установка соединения с сервером,и во-вторых, не требуется пересылка лишнего по сети. PL/pgSQL очень удобен для таких задач, поскольку тесно интегрирован с SQL. Подпрограммы на других языках могут пользоваться специальным интерфейсом SPI для работы с базой данных, в контексте подключения к которой они работают.

Ко второй группе можно отнести любые вычисления, не связанные с обращением к базе данных. Здесь возможности PL/pgSQL сильно ограничены. Начать с того, что он вычислительно не эффективен: любое выражение вычисляется с помощью запроса (хоть в PostgreSQL версии 13 эта ситуация и улучшена).

Если бы были важны только эффективность и универсальность, можно было бы использовать язык C. Но писать на нем прикладной код крайне дорого и долго.

Кроме того, язык PL/pgSQL достаточно старомоден и не располагает возможностями и библиотеками, которые есть в современных ЯП, и он может быть в принципе не пригоден для решения целого ряда задач.

Поэтому использование других языков (отличных от SQL, PL/pgSQL и C) для серверного программирования может быть вполне оправдано.

Интерфейс SPI

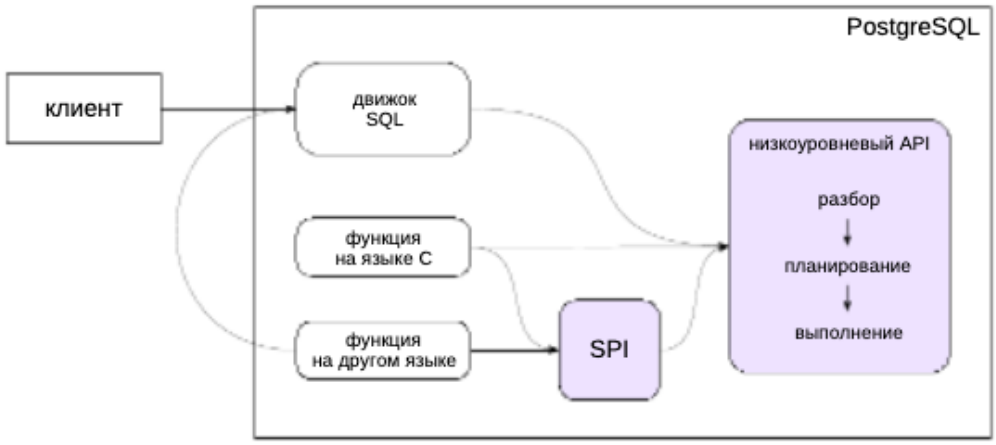

Рисунок 2 – обработка обращения к БД

PostgreSQL располагает внутренним низкоуровневым API для работыс базой данных. Сюда входят функции для разбора и переписывания запроса, построения плана, выполнения запроса (включая обращенияк таблицам, индексам и т. п.). Это самый эффективный способ работыс данными, но он требует внимания ко многим деталям, таким, как управление памятью, установка блокировок и т. д.

Движок SQL, являющийся частью каждого обслуживающего процесса, пользуется именно этим, низкоуровневым, API.

Для удобства имеется более высокоуровневый (но в ряде случаев менее эффективный) интерфейс: SPI, Server Programming Interface.

Интерфейс программирования сервера (SPI, Server Programming Interface) даёт разработчикам пользовательских функций на C возможность запускать команды SQL из своих функций или процедур. SPI представляет собой набор интерфейсных функций, упрощающих доступ к анализатору, планировщику и исполнителю запросов. В SPI есть также функции для управления памятью.

Интерфейс SPI рассчитан на язык C: подпрограмма на C может пользоваться как этим интерфейсом, так и – при необходимости – низкоуровневым.

Но, как правило, авторы языков программирования предоставляют

обертки, позволяющие подпрограммам на этих языках тоже пользоваться SPI. Это значительно выгоднее, чем инициировать из подпрограммы новое соединение и использовать для работы с базой возможность клиент-серверного протокола.

Практика 3. Выполняется на вашей БД по аналогии нижеприведенной.

Для доступа к возможностям SPI подпрограммы на языке Python автоматически импортируют модуль plpy (мы уже использовали функцию info из этого модуля — аналог команды RAISE INFO языка PL/pgSQL).

```
=> CREATE TABLE test(
    n integer PRIMARY KEY,
    descr text
);
CREATE TABLE
=> INSERT INTO test VALUES (1,'foo'), (2,'bar'), (3,'baz');
INSERT 0 3
```

Напишем для примера функцию, возвращающую текстовое описание по ключу. Отсутствие ключа должно приводить к ошибке.

Какие конструкции здесь соответствуют обращению к SPI?

```
=> CREATE OR REPLACE FUNCTION get_descr_py(n integer) RETURNS
text
AS $python$
    if "plan_get_descr_py" in SD:
        plan = SD["plan_get_descr_py"]
    else:
        plan = plpy.prepare(
            "SELECT descr FROM test WHERE n = $1", ["integer"]
        )
        SD["plan_get_descr_py"] = plan
    rows = plan.execute([n])
    if rows.nrows() == 0:
```

```
        raise plpy.spiexceptions.NoDataFound()
    else:
        return rows[0]["descr"]
$python$ LANGUAGE plpython3u STABLE;
CREATE FUNCTION
```

Вызов plpy.prepare соответствует функции SPI_prepare (и SPI_keepplan), а plpy.execute — функции SPI_execute_plan. Также неявно вызывается SPI_connect и SPI_finish. То есть обертка языка может дополнительно упрощать интерфейс.

Обратите внимание:

Чтобы сохранить план подготовленного запроса, приходится использовать словарь SD, сохраняемый между вызовами функции;

Требуется явная проверка того, что строка была найдена.

```
=> SELECT get_descr_py(1);
 get_descr_py
--------------
 foo
(1 row)

=> SELECT get_descr_py(42);
ERROR:  spiexceptions.NoDataFound:
CONTEXT:  Traceback (most recent call last):
  PL/Python function "get_descr_py", line 11, in <module>
    raise plpy.spiexceptions.NoDataFound()
PL/Python function "get_descr_py"
```

Показательно сравнить с аналогичной процедурой на языке PL/pgSQL:

```
=> CREATE OR REPLACE FUNCTION get_descr(n integer) RETURNS text
AS $$
DECLARE
    descr text;
BEGIN
    SELECT t.descr INTO STRICT descr
    FROM test t WHERE t.n = get_descr.n;
```

```
    RETURN descr;
END;
$$ LANGUAGE plpgsql STABLE;
CREATE FUNCTION
```

План подготавливается и переиспользуется автоматически;

Проверка существования строки указывается словом STRICT.

```
=> SELECT get_descr(1);
 get_descr
-----------
 foo
(1 row)

=> SELECT get_descr(42);
ERROR:  query returned no rows
CONTEXT:  PL/pgSQL function get_descr(integer) line 5 at SQL
statement
```

Другие языки могут предоставлять другой способ доступа к функциям SPI, а могут и не предоставлять.

Из чего выбирать

Интенсивная работа с данными

- SQL, PL/pgSQL

Проверенные, часто используемые

- PL/Perl, PL/Python, PL/V8, PL/Java

Другие языки общего назначения

- PL/Go, PL/Lua, ...

Специальные задачи

- PL/R (статистика), PL/Proxy (удаленные вызовы), ...

Максимальная эффективность

- C

Если хранимые процедуры используются для интенсивной работы с данными, лучшим выбором скорее всего будут обычные SQL и PL/pgSQL.

Для других задач из штатных языков часто используют PL/Perl и PL/Python, а из сторонних — PL/V8 и PL/Java. (С PL/V8 могут быть сложности

из-за того, что Debian и Red Hat перестали поставлять это расширение в виде пакета из-за сложностей со сборкой.)

Можно попробовать и другие языки, например, PL/Go или PL/Lua, если в этом есть смысл. Обязательно обращайте внимание на состояние проекта, активность сообщества, возможность получить поддержку.

Есть ряд специализированных языков. Например, для статистической обработки данных пригодится PL/R, для удаленного вызова процедур и шардинга может использоваться PL/Proxy.

Не следует сбрасывать со счетов и язык C, если нужна максимальная эффективность. Безусловно, это сложнее, но в документации ив исходном коде PostgreSQL есть множество примеров, на основе которых можно писать собственные функции.

Практика 4. Выполняется на вашей БД по аналогии нижеприведенной.

```
student$ psql bookstore2
```

Трансформация

```
=> CREATE EXTENSION jsonb_plpython3u;
CREATE EXTENSION

CREATE FUNCTION empapi.run(task public.tasks) RETURNS text
    LANGUAGE plpython3u
    AS $_$
    # получаем параметры задания
    p = plpy.prepare("SELECT func FROM empapi.get_programs()
WHERE program_id = $1", ["bigint"])
    r = p.execute([task["program_id"]])
    func = r[0]["func"]
    # выполняем функцию и получаем результат
    p = plpy.prepare("SELECT * FROM " + plpy.quote_ident(func) +
"($1)", ["jsonb"])
    r = p.execute([task["params"]])
    # вычисляем максимальную ширину каждого столбца
    cols = r.colnames()
```

```
    collen = {col: len(col) for col in cols}
    for i in range(len(r)):
        for col in cols:
            if len( str(r[i][col]) ) > collen[col]:
                collen[col] = len( str(r[i][col]) )
    # выводим названия столбцов с учетом ширины
    res = ""
    res += " ".join( [col.center(collen[col]," ") for col in
cols] ) + "\n"
    # отбивка из минусов
    res += " ".join( ["-"*collen[col] for col in cols] ) + "\n"
    # выводим результат
    for i in range(len(r)):
        res += " ".join( [str(r[i][col]).ljust(collen[col]," ")
for col in cols] ) + "\n"
    return res
$_$;
```

Поскольку добавить трансформацию в существующую функцию командой ALTER невозможно, пересоздадим функцию без изменений, добавив предложение TRANSFORM.

```
=> CREATE OR REPLACE FUNCTION public.run(func text, params
jsonb)
RETURNS text
TRANSFORM FOR TYPE jsonb
AS $python$
    # выполняем функцию и получаем результат
    p = plpy.prepare("SELECT * FROM " + plpy.quote_ident(func) +
"($1)", ["jsonb"])
    r = p.execute([params])
    # вычисляем максимальную ширину каждого столбца
    cols = r.colnames()
    collen = {col: len(col) for col in cols}
    for i in range(len(r)):
        for col in cols:
```

```
            if len( str(r[i][col]) ) > collen[col]:
                collen[col] = len( str(r[i][col]) )
    # выводим названия столбцов с учетом ширины
    res = ""
    res += " ".join( [col.center(collen[col]," ") for col in
cols]) + "\n"
    # отбивка из минусов
    res += " ".join( ["-"*collen[col] for col in cols]) + "\n"
    # выводим результат
    for i in range(len(r)):
        res += " ".join( [str(r[i][col]).ljust(collen[col]," ")
for col in cols]) + "\n"
    return res
$python$ LANGUAGE plpython3u VOLATILE;
CREATE FUNCTION
```

Почтовые сообщения

Простая функция для отправки почтовых сообщений через локальный почтовый сервер может выглядеть так:

```
CREATE OR REPLACE FUNCTION public.sendmail(from_addr text,
to_addr text, subj text, msg text)
 RETURNS void
 LANGUAGE plpython3u
AS $function$
    import smtplib
    server = smtplib.SMTP('localhost')
    server.sendmail(
        from_addr,
        to_addr,
        "\r\n".join([
            "From: %s" % from_addr,
            "To: %s" % to_addr,
            "Content-Type: text/plain; charset=\"UTF-8\"",
            "Subject: %s" % subj,
            "\r\n%s" % msg
        ]).encode('utf-8')
```

```
    )
    server.quit()
$function$;
```

Модуль email дает больше возможностей, но они нам не нужны.

Создадим фоновое задание для отправки писем:

```
create table programs (program_id bigint, name text, func text);

  CREATE OR REPLACE FUNCTION public.register_program(name text,
func text)
        RETURNS bigint
        LANGUAGE sql
          AS $function$
           INSERT INTO programs(name, func) VALUES (name, func)
          RETURNING program_id;
        $function$;

=> CREATE FUNCTION public.sendmail_task(params jsonb)
RETURNS text
AS $$
    SELECT sendmail(
        from_addr => params->>'from_addr',
        to_addr   => params->>'to_addr',
        subj      => params->>'subj',
        msg       => params->>'msg'
    );
    SELECT 'OK';
$$ LANGUAGE sql VOLATILE;
CREATE FUNCTION
=> SELECT register_program('Отправка письма', 'sendmail_task');
 register_program
------------------
                2
(1 row)
```

Функция checkout книжного приложения содержит вызов дополнительной функции, куда мы и поместим логику отправки письма:

```
CREATE TABLE public.tasks (
    task_id bigint NOT NULL,
    program_id bigint NOT NULL,
    status text DEFAULT 'scheduled'::text NOT NULL,
    params jsonb,
    pid integer,
    started timestamp with time zone,
    finished timestamp with time zone,
    result text,
    host text,
    port text,
    CONSTRAINT tasks_check CHECK ((((host IS NOT NULL) AND (port
IS NOT NULL)) OR ((host IS NULL) AND (port IS NULL)))),
    CONSTRAINT tasks_status_check CHECK ((status = ANY
(ARRAY['scheduled'::text, 'running'::text, 'finished'::text,
'error'::text])))
);

CREATE FUNCTION public.run_program(program_id bigint, params
jsonb DEFAULT NULL::jsonb, host text DEFAULT NULL::text, port
text DEFAULT NULL::text) RETURNS bigint
    LANGUAGE sql SECURITY DEFINER
    AS $$
    INSERT INTO tasks(program_id, status, params, host, port)
    VALUES (program_id, 'scheduled', params, host, port)
    RETURNING task_id;
$$;

=> CREATE OR REPLACE FUNCTION public.before_checkout(user_id
bigint)
RETURNS void
AS $$
```

```
<<local>>
DECLARE
    params jsonb;
BEGIN
    SELECT jsonb_build_object(
        'from_addr', 'bookstore@localhost',
        'to_addr',    u.email,
        'subj',       'Поздравляем с покупкой',
        'msg',        format(
                          E'Уважаемый %s!\nВы совершили покупку
на общую сумму %s ₽.',
                          u.username,
                          '1000'
                      )
    )
    INTO params
    FROM users u
        JOIN cart_items ci ON ci.user_id = u.user_id
    WHERE u.user_id = before_checkout.user_id
    GROUP BY u.user_id;

    PERFORM public.run_program(
        program_id => 2,
        params => params
    );
END;
$$ LANGUAGE plpgsql VOLATILE;
CREATE FUNCTION
```

Итоги

PostgreSQL позволяет подключать любые языки.

Богатый выбор языков программирования позволяет решать любые задачи на стороне сервера.

Задание на практическую работу:

1. Функция empapi.run, написанная на языке Python, принимает параметр типа jsonb. Добавьте трансформацию, чтобы избежать

преобразований в текстовый вид и обратно.

2. Отправляйте пользователю, совершившему покупку, письмо-подтверждение с указанием суммы.
   В PostgreSQL нет встроенной функции для отправки писем, но ее можно реализовать на каком-либо недоверенном языке. Убедитесь, что письмо не может быть отправлено до того, как транзакция покупки будет завершена.
   Для отправки пользуйтесь уже готовой функцией public.sendmail (посмотрите ее определение) или напишите свою. Посылать письмо внутри транзакции покупки неправильно: транзакция может быть оборвана по какой-либо причине, а письмо уже уйдет.
   Воспользуйтесь механизмом фоновых заданий: в транзакции добавляйте задание на отправку письма. В таком случае оно будет отправлено, только если транзакция завершится успешно.