

Расширяемость Языки программирования



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.

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

«Встроенные»

- C
- SQL

Стандартные (поддержка сообщества)

- PL/pgSQL
- PL/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.

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 (только Python3, поддержка Python2 прекращена) 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. Название несколько неудачное: никто не мешает подключить к PostgreSQL *функциональный* или еще какой-нибудь язык. Поэтому лучше понимать PL как programming language, а не как procedural language.

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

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

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

ЯП разделяются на доверенные (trusted) и недоверенные (untrusted).

Доверенные языки должны соблюдать ограничения доступа, установленные в системе. Если пользователь не имеет доступа к данным, подпрограмма на доверенном языке не должна предоставить ему такой доступ.

По сути это означает, что язык должен ограничивать взаимодействие пользователя с окружением (например, с операционной системой) и с 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, в том числе с указанием SECURITY DEFINER, и выдать право на ее исполнение обычным пользователям.

Языки программирования

```
=> 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;
```

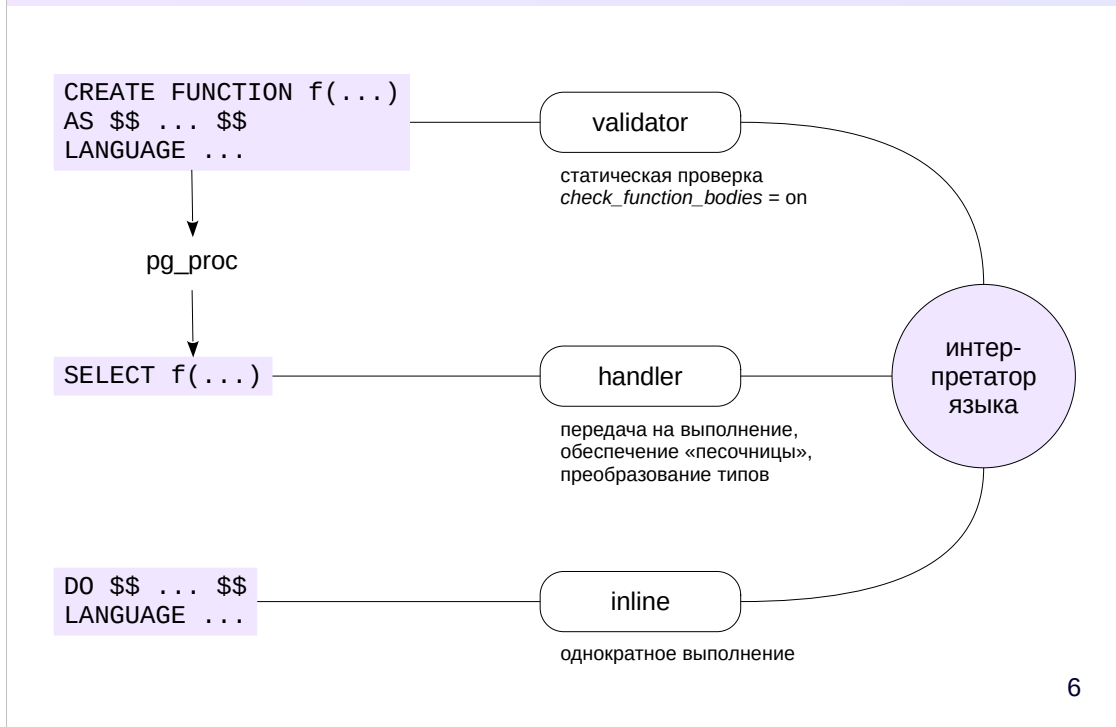
```
$perl$ LANGUAGE plperl VOLATILE;
```

```
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



6

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

При создании хранимой подпрограммы, ее код проверяется на наличие ошибок (**validator**). Если проверка проходит успешно, подпрограмма сохраняется в системном каталоге. При этом если подпрограмма оформлена в стиле стандарта SQL, то ее тело хранится в виде дерева разбора и будет обрабатываться обычным образом. В остальных случаях сохраняется только исходный код в виде текстовой строки. Компиляция поддерживается только для подпрограмм на языке C.

Проверку можно отключить параметром `check_function_bodies`; утилита `pg_dump` пользуется этим, чтобы снять зависимость от порядка создания объектов.

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

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

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

Из этих трех функций обязательна только функция `handler`.

Подключение нового языка

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

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

А для доверенного (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.

Трансформации типов

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;
```

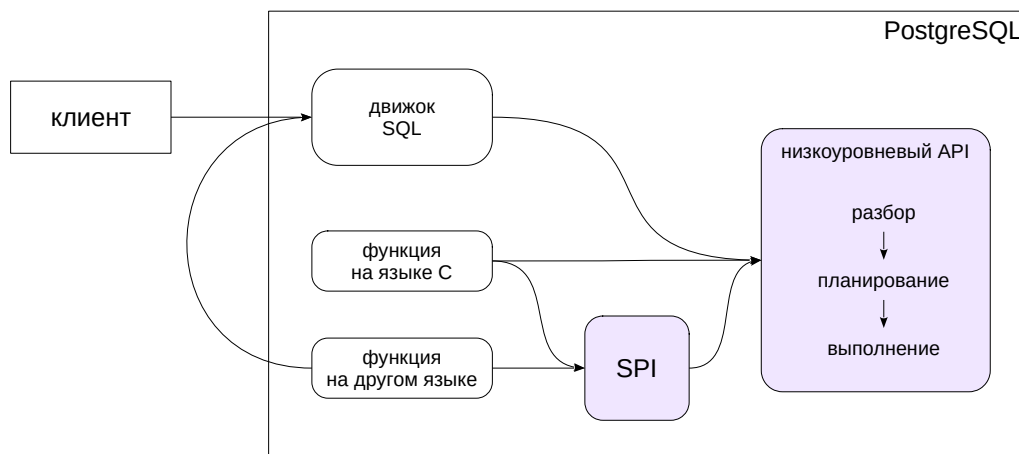
```
CREATE FUNCTION
```

```
=> 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).

Интерфейс SPI



PostgreSQL располагает внутренним низкоуровневым API для работы с базой данных. Сюда входят функции для разбора и переписывания запроса, построения плана, выполнения запроса (включая обращения к таблицам, индексам и т. п.). Это самый эффективный способ работы с 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.

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

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 клиент-серверный протокол.

Интерфейс SPI

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 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 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.

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.

Ко второй группе можно отнести обработку, не связанную 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 (для которого в ряде простых случаев исключен этап планирования).

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 в принципе не пригоден для решения целого ряда задач.

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

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/Proху (удаленные вызовы), ...

Максимальная эффективность

C

Если подпрограммы используются для интенсивной работы с данными, лучшим выбором вероятно 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 обращайтесь 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 можно использовать PL/Proху.

В документации упомянуто некоторое количество языков:

<https://postgrespro.ru/docs/postgresql/16/external-pl>

Больше информации о доступных языках есть на вики:

https://wiki.postgresql.org/wiki/PL_Matrix

Но и это не полный список. Есть множество других проектов, хотя многие из них не продвигаются дальше первой версии. Не следует сбрасывать со счетов и язык C, если нужна максимальная эффективность. Это сложнее, но в документации и в исходном коде PostgreSQL есть множество соответствующих примеров.

Пример специализированного языка: XSLT

В качестве иллюстрации очень специализированного языка (который никак нельзя назвать процедурным!) посмотрим на PL/XSLT. Допустим, мы получаем данные для отчета с помощью запроса, и представляем их в формате XML:

```
=> SELECT *
FROM query_to_xml('SELECT n, descr FROM test',true,false,'');

-----
query_to_xml
-----
<table xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">+
  <row>+
    <n>1</n>+
    <descr>foo</descr>+
  </row>+
  <row>+
    <n>2</n>+
    <descr>bar</descr>+
  </row>+
  <row>+
    <n>3</n>+
    <descr>baz</descr>+
  </row>+
</table>+

(1 row)
```

Чтобы вывести отчет пользователю, его можно преобразовать в формат HTML с помощью XSLT-преобразования. Подключим язык PL/XSLT:

```
=> CREATE EXTENSION plxslt;
```

```
CREATE EXTENSION
```

Вот так может выглядеть простое преобразование:

```
=> CREATE FUNCTION html_report(xml) RETURNS xml
AS $xml$
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html><body><table>
  <xsl:for-each select="table/row">
    <tr>
      <td><xsl:value-of select="n"/></td>
      <td><xsl:value-of select="descr"/></td>
    </tr>
  </xsl:for-each>
</table></body></html>
</xsl:template>
</xsl:stylesheet>
$xml$ LANGUAGE xslt IMMUTABLE;
```

```
CREATE FUNCTION
```

И вот результат:

```
=> SELECT * FROM html_report(
  query_to_xml('SELECT * FROM test',true,false,'')
);
```

```
      html_report
-----
<html><body><table>  +
<tr>                +
<td>1</td>          +
<td>foo</td>        +
</tr>               +
<tr>                +
<td>2</td>          +
<td>bar</td>        +
</tr>               +
<tr>                +
<td>3</td>          +
<td>baz</td>        +
</tr>               +
</table></body></html>+

(1 row)
```

Таким образом можно разделить логику отчета (обычный SQL-запрос) и его представление.

PostgreSQL позволяет подключать любые языки

Богатый выбор языков программирования позволяет решать любые задачи на стороне сервера



1. Отправляйте пользователю, совершившему покупку в магазине, письмо-подтверждение с указанием суммы. В виртуальной машине настроен локальный почтовый сервер, пересылающий любую исходящую почту в локальный ящик пользователя student. 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.

1. 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.

1. Почтовые сообщения

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

```
=> \sf sendmail
```

```
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 FUNCTION public.sendmail_task(params jsonb) RETURNS text
  LANGUAGE sql VOLATILE
  BEGIN ATOMIC
    SELECT sendmail(
        from_addr => params->>'from_addr',
        to_addr   => params->>'to_addr',
        subj      => params->>'subj',
        msg       => params->>'msg'
    );
    SELECT 'OK';
  END;

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 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,
            sum(ci.qty * get_retail_price(ci.book_id))
        )
    )
    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 empapi.run_program(
        program_id => 2,
        params => params
    );
END;
$$ LANGUAGE plpgsql VOLATILE;

CREATE FUNCTION

```

1. Языки Python и Perl имеют удобный тип данных — ассоциативный массив, — который отсутствует в PL/pgSQL. В Python это dict (словарь), в Perl — hash (хеш-таблица).
Напишите на одном из этих языков функцию, получающую на вход текстовую строку и возвращающую таблицу из слов, которые встречаются в этой строке, с указанием количества вхождений.
Решается ли эта задача на языке SQL?
2. Напишите функцию, которая по имени файла определяет и выводит его MIME-тип, аналогично команде shell `file --brief --mime-type`

15

1. Например:

```
SELECT * FROM words_count('the best of the best');
 word | cnt
-----+-----
 the  |    2
 of   |    1
 best |    2
```

2. Воспользуйтесь расширением plsh, установленным в виртуальной машине курса. Оно предоставляет язык PL/sh для написания функций на языке командой оболочки Unix.

Пример использования, приведенный автором (Питер Эйзен траут):

```
CREATE FUNCTION concat(text, text) RETURNS text AS $$
#!/bin/sh
echo "$1$2"
$$ LANGUAGE plsh;
```

Обратите внимание, что обращение к параметрам возможно только по номеру.

1. Количество вхождений слов в строку

```
=> CREATE DATABASE ext_languages;
```

```
CREATE DATABASE
```

```
=> \c ext_languages
```

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

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

```
CREATE EXTENSION
```

```
=> CREATE FUNCTION words_count(s text)
RETURNS TABLE(word text, cnt integer)
```

```
AS $python$
```

```
    words = {}
```

```
    for w in s.split():
```

```
        words[w] = words.get(w, 0) + 1
```

```
    return words.items()
```

```
$python$ LANGUAGE plpython3u IMMUTABLE;
```

```
CREATE FUNCTION
```

```
=> SELECT * FROM words_count('раз два три два три три');
```

```
word | cnt
-----+-----
раз  |    1
два  |    2
три  |    3
(3 rows)
```

На SQL задача тоже решается элементарно. Здесь вместо явного использования ассоциативного массива мы полагаемся на реализацию группировки:

```
=> SELECT word, count(*)
```

```
FROM regexp_split_to_table('раз два три два три три', '\s+') word
```

```
GROUP BY word;
```

```
word | count
-----+-----
три  |     3
раз  |     1
два  |     2
(3 rows)
```

Фактически в обоих случаях используется хеш-таблица:

```
=> EXPLAIN (costs off) SELECT word, count(*)
```

```
FROM regexp_split_to_table('раз два три два три три', '\s+') word
```

```
GROUP BY word;
```

QUERY PLAN

```
-----
HashAggregate
  Group Key: word
  -> Function Scan on regexp_split_to_table word
(3 rows)
```

2. Тип файла

```
=> CREATE EXTENSION plsh;
```

```
CREATE EXTENSION
```

```
=> CREATE FUNCTION file_type(file text) RETURNS text AS $$
```

```
#!/bin/bash
```

```
file --brief --mime-type $1
```

```
$$ LANGUAGE plsh VOLATILE;
```

```
CREATE FUNCTION
```

```
=> SELECT file_type('/home/student/covers/novikov_dbtech2.jpg');
```

```
file_type
-----
image/jpeg
(1 row)
```