

## Введение в функции PostgreSQL (Function)

Инструкция **create function** позволяет определить новую пользовательскую функцию.

### Синтаксис инструкции create function:

```
create [or replace] function function_name(param_list)
    returns return_type
    language plpgsql
as
$$
declare
-- variable declaration
begin
-- logic
end;
$$
```

Язык кода: диалект SQL PostgreSQL и PL / pgSQL (pgsql)

В этом синтаксисе:

- Сначала укажите имя функции после ключевых слов **create function**. Если вы хотите заменить существующую функцию, вы можете использовать ключевые слова **or replace**.
- Затем укажите список параметров функции, заключенный в круглые скобки после имени функции. Функция может иметь ноль или много параметров.
- Затем укажите тип данных возвращаемого значения после ключевого слова **returns**.
- После этого используйте **language plpgsql**, чтобы указать процедурный язык функции. Обратите внимание, что PostgreSQL поддерживает многие процедурные языки, а не только **plpgsql**.
- Наконец, поместите блок в строковую константу, заключенную в долларовые кавычки.

### Синтаксис полного блока в PL/ pgSQL:

```
[ <<label>> ]
[ declare
    declarations ]
begin
    statements;
...
end [ label ];
```

PL / pgSQL поддерживает три режима параметров: **in**, **out**, и **inout**. По умолчанию параметр принимает режим **in**.

- Используйте режим **in**, если вы хотите передать значение функции.
- Используйте режим **out**, если вы хотите вернуть значение из функции.
- Используйте режим **inout**, когда вы хотите передать начальное значение, обновить значение в функции и вернуть ему обновленное значение.

**Пример 1:** функция **swap** принимает два целых числа и их значения:

```

create or replace function swap(
  inout x int,
  inout y int
)
language plpgsql
as $$
begin
  select x,y into y,x;
end; $$;

```

Следующий оператор вызывает `swap()` функцию:

```
select * from swap(10,20);
```

	x integer	y integer
1	20	10

Мы будем использовать таблицу `film` из базы данных примеров `dvdrental`.

film
* film_id title description release_year language_id rental_duration rental_rate length replacement_cost rating last_update special_features fulltext

### **Пример 2:**

Следующий оператор создает функцию, которая подсчитывает фильмы, длина которых между параметрами `len_from` и `len_to`:

```

create function get_film_count(len_from int, len_to int)
returns int
language plpgsql
as
$$
declare
  film_count integer;
begin
  select count(*)
  into film_count
  from film
  where length between len_from and len_to;

```

```

    return film_count;
end;
$$;

```

Функция `get_film_count` состоит из двух основных разделов: заголовка и тела.

В разделе заголовка:

- Во-первых, имя функции `get_film_count` следует за ключевыми словами `create function`.
- Во-вторых, функция `get_film_count()` принимает два параметра `len_from` и `len_to` с целочисленным типом данных.
- В-третьих, функция `get_film_count` возвращает целое число, указанное в предложении `returns int`.
- Наконец, язык функции `plpgsql` обозначается символом `language plpgsql`.

В теле функции:

- Используйте синтаксис строковой константы, заключенной в долларские кавычки, которая начинается `$$` и заканчивается `$$`. Между ними `$$` вы можете поместить **блок**, содержащий объявление и логику функции.
- В разделе объявления объявите переменную с именем `film_count`, в которой хранится количество фильмов, выбранных из таблицы `film`.
- В теле блока используйте инструкцию `select into`, чтобы выбрать количество фильмов, длина которых находится между `len_from` и `len_to` и присвоить результат переменной `film_count`. В конце блока используйте инструкцию `return` для возврата `film_count`.

### Пример 3:

В следующем примере определяется функция `get_film_stat`, которая имеет три параметра `out`:

```

create or replace function get_film_stat(
    out min_len int,
    out max_len int,
    out avg_len numeric)
language plpgsql
as $$
begin
    select min(length),
           max(length),
           avg(length)::numeric(5,1)
    into min_len, max_len, avg_len
    from film;
end;$$

```

В функции `get_film_stat` мы выбираем минимальное, максимальное и среднее значение длины пленки из таблицы `film` с помощью агрегатных функций `min`, `max`, и `avg` и присваиваем результаты соответствующим параметрам `out`.

Следующий оператор вызывает функцию `get_film_stat`:

```

select get_film_stat();

```

	get_film_stat record
1	(46,185,115.3)

Результатом функции является запись. Чтобы выходные данные были разделены в виде столбцов, вы используете следующую инструкцию:

```
select * from get_film_stat();
```

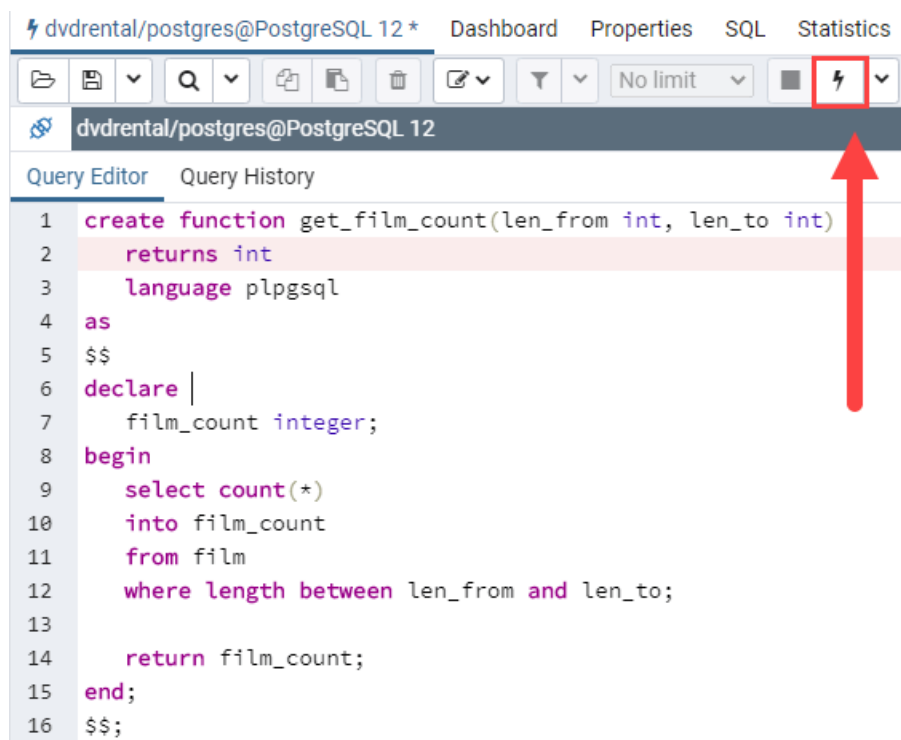
	min_len integer	max_len integer	avg_len numeric
1	46	185	115.3

### Создание функции с помощью pgAdmin

Сначала запустите инструмент pgAdmin и подключитесь к базе данных образцов dvdrental.

Во-вторых, откройте инструмент запроса, выбрав **Инструменты > Инструмент запроса**.

В-третьих, введите приведенный выше код в инструмент запроса и нажмите кнопку **Выполнить**, чтобы создать функцию get\_film\_count.



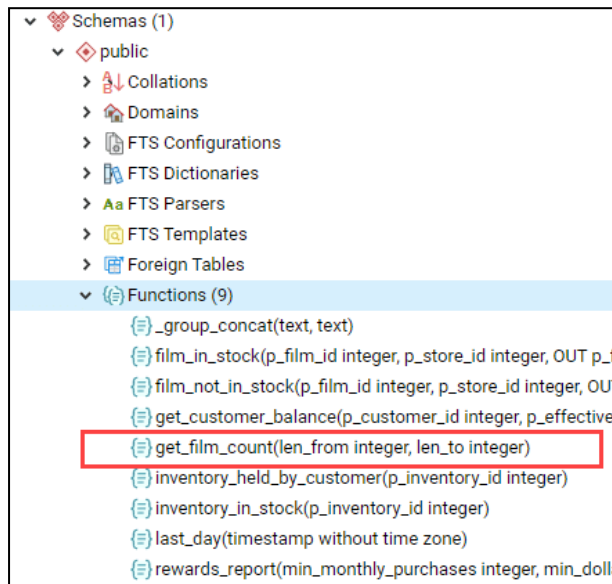
Если все в порядке, вы увидите следующее сообщение:

**CREATE FUNCTION**

Query returned successfully in 44 msec.

Это означает, что функция get\_film\_count создана успешно.

Наконец, вы можете найти функцию get\_film\_count в списке **функций**:



В случае, если вы не смогли найти имя функции, вы можете щелкнуть правой кнопкой мыши узел **Функции** и выбрать пункт меню **Обновить ...**, чтобы обновить список функций.

## PostgreSQL – введение в хранимые процедуры

PostgreSQL позволяет пользователям расширять функциональность базы данных с помощью определяемых пользователем функций и хранимых процедур с помощью различных процедурных языковых элементов, которые часто называют хранимыми процедурами.

Процедуры хранилища определяют функции для создания триггеров или пользовательских агрегатных функций. Кроме того, хранимые процедуры также добавляют множество процедурных функций, например, структуры управления и сложные вычисления. Это позволяет вам разрабатывать пользовательские функции намного проще и эффективнее.

PostgreSQL классифицирует процедурные языки на две основные группы:

1. Безопасные языки могут использоваться любыми пользователями. SQL и PL / pgSQL являются безопасными языками.
2. Изолированные языки используются только суперпользователями, поскольку изолированные языки предоставляют возможность обхода безопасности и разрешают доступ к внешним источникам. С является примером изолированного языка.

По умолчанию PostgreSQL поддерживает три процедурных языка: SQL, PL / pgSQL и C. Вы также можете загрузить в PostgreSQL другие процедурные языки, например, Perl, Python и TCL, используя расширения.

### Преимущества использования хранимых процедур PostgreSQL:

Хранимые процедуры имеют много преимуществ, а именно:

- Сократите количество циклов между приложениями и серверами баз данных. Все инструкции SQL заключены в функцию, хранящуюся на сервере базы данных PostgreSQL, поэтому приложению достаточно выполнить вызов функции, чтобы получить результат обратно, вместо отправки нескольких инструкций SQL и ожидания результата между каждым вызовом.
- Повышение производительности приложения, поскольку определяемые пользователем функции и хранимые процедуры предварительно компилируются и хранятся на сервере базы данных PostgreSQL.
- Повторно используемый во многих приложениях. После разработки функции вы можете повторно использовать ее в любых приложениях.

## ОСНОВЫ

Чтобы определить новую хранимую процедуру, используете инструкцию `create procedure`.

Ниже приведен базовый синтаксис инструкции `create procedure`:

```
create [or replace] procedure procedure_name(parameter_list)
language plpgsql
as $$
declare
-- variable declaration
begin
-- stored procedure body
end; $$
```

В этом синтаксисе:

- Во-первых, укажите имя хранимой процедуры после ключевых слов `create procedure`.
- Во-вторых, определите параметры для хранимой процедуры. Хранимая процедура может принимать ноль или более параметров.
- В-третьих, укажите `plpgsql` в качестве процедурного языка хранимую процедуру. Обратите внимание, что вы можете использовать другие процедурные языки для хранимой процедуры, такие как SQL, C и т.д.
- Наконец, используйте синтаксис строковой константы, заключенной в долларские кавычки, для определения тела хранимой процедуры.

Параметры в хранимых процедурах могут иметь режимы `in` и `inout` У них не может быть такого `out` режима.

Хранимая процедура не возвращает значения. Вы не можете использовать оператор `return` со значением внутри процедуры, подобной этой:

```
return expression;
```

Однако вы можете использовать `return` инструкцию без команды `expression` немедленно остановить хранимую процедуру:

```
return;
```

Если вы хотите вернуть значение из хранимой процедуры, вы можете использовать параметры с режимом `inout`.

## Пример:

Для демонстрации мы будем использовать следующую таблицу учетных записей:  
`drop table if exists accounts;`

```
create table accounts (
  id int generated by default as identity,
  name varchar(100) not null,
  balance dec(15, 2) not null,
  primary key(id)
);
```

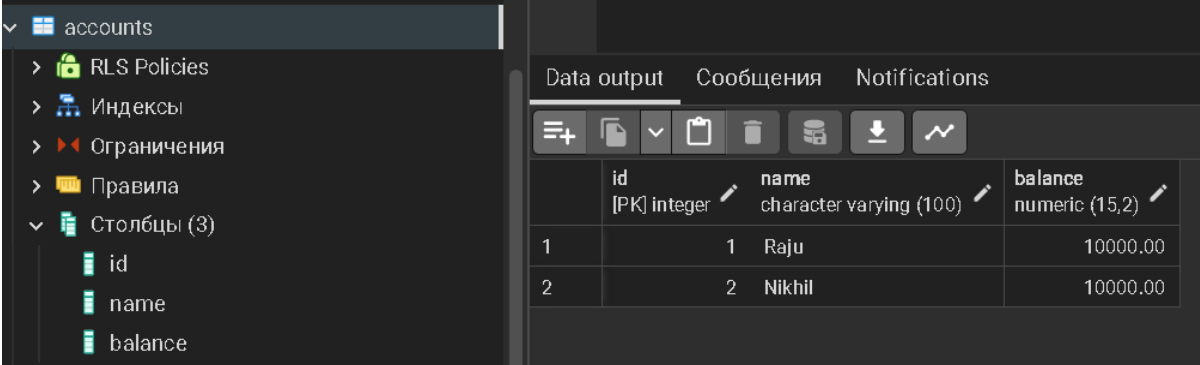
```
insert into accounts(name, balance)
values('Raju', 10000);
insert into accounts(name, balance)
```

```
values('Nikhil', 10000);
```

Следующий запрос покажет данные таблицы:

```
select * from accounts;
```

Это изображает результат, как показано ниже:



	id [PK] integer	name character varying (100)	balance numeric (15,2)
1	1	Raju	10000.00
2	2	Nikhil	10000.00

Следующий запрос создает хранимую процедуру с именем transfer, которая переводит указанную сумму денег с одного счета на другой.

```
create or replace procedure transfer(
  sender int,
  receiver int,
  amount dec
)
language plpgsql
as $$
begin
  -- subtracting the amount from the sender's account
  update accounts
  set balance = balance - amount
  where id = sender;
  -- adding the amount to the receiver's account
  update accounts
  set balance = balance + amount
  where id = receiver;
  commit;
end;$$;
```

### Вызов хранимой процедуры

Чтобы вызвать хранимую процедуру, вы используете оператор CALL следующим образом:

```
call stored_procedure_name(argument_list);
```

### Пример:

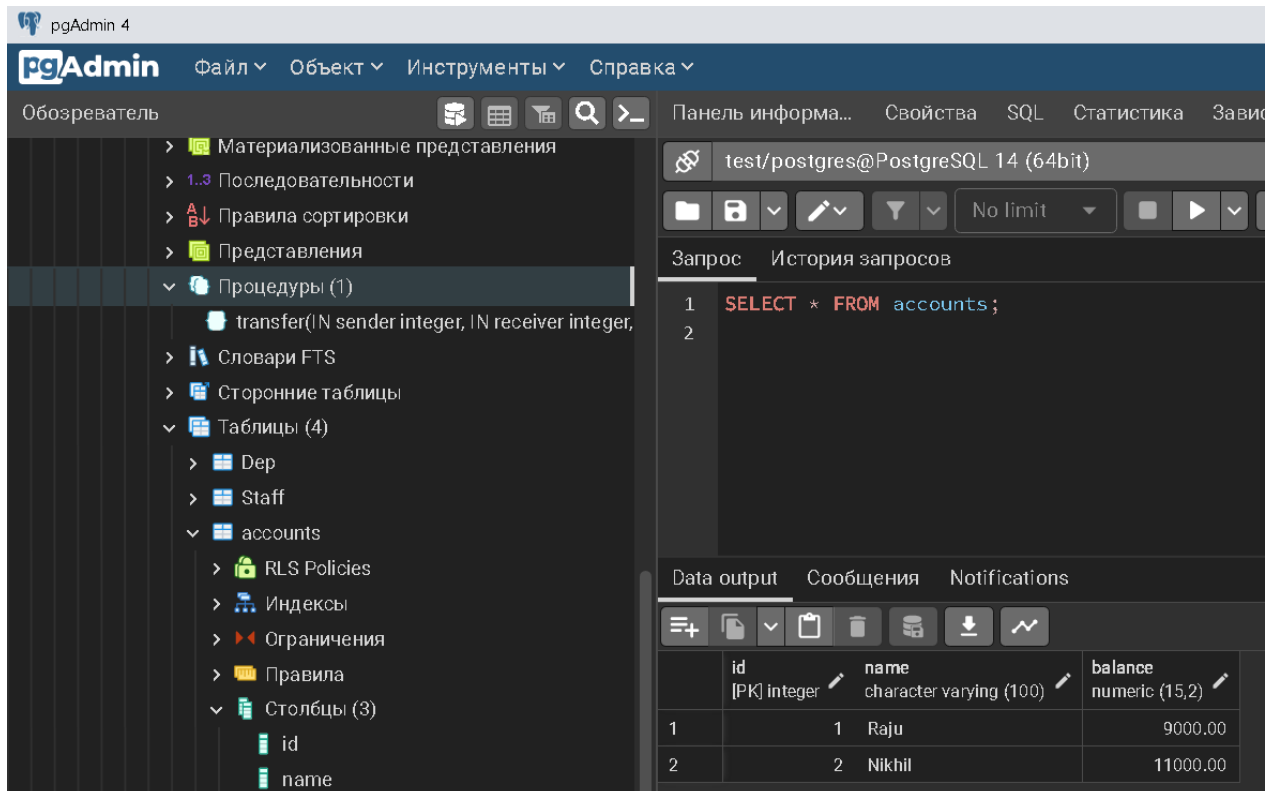
Приведенная ниже инструкция вызывает хранимую процедуру transfer для перевода 1000 долларов США со счета Раджу на счет Нихила:

```
call transfer(1, 2, 1000);
```

Следующая инструкция проверяет данные в таблице accounts после переноса:

```
SELECT * FROM accounts;
```

### Вывод:



## Создание триггеров в PostgreSQL

Триггер определяет операцию, которая должна выполняться при наступлении некоторого события в базе данных. Триггеры срабатывают при выполнении с таблицей команды SQL INSERT, UPDATE или DELETE.

В PostgreSQL триггеры создаются на основе существующих функции, т.е. сначала командой CREATE FUNCTION определяется триггерная функция, затем на ее основе командой CREATE TRIGGER определяется собственно триггер.

## Синтаксис определения триггера

**CREATE TRIGGER триггер**

{ BEFORE | AFTER } { *событие* [ OR событие ] } ON таблица

FOR EACH { ROW | STATEMENT }

EXECUTE PROCEDURE функция ( аргументы )

**CREATE TRIGGER триггер.** В аргументе *триггер* указывается произвольное имя создаваемого триггера. Имя может совпадать с именем триггера, уже существующего в базе данных при условии, что этот триггер установлен для другой таблицы. Кроме того, по аналогии с большинством других несистемных объектов баз данных, имя триггера (в сочетании с таблицей, для которой он устанавливается) должно быть уникальным лишь в контексте базы данных, в которой он создается

**{ BEFORE | AFTER }.** Ключевое слово BEFORE означает, что функция должна выполняться *перед* попыткой выполнения операции, включая все встроенные проверки ограничений данных, реализуемые при выполнении команд INSERT и DELETE. Ключевое слово AFTER означает, что функция вызывается после завершения операции, приводящей в действие триггер.



{ *событие* [ **OR событие** ... ] }. События SQL, поддерживаемые в PostgreSQL: INSERT, UPDATE или DELETE. При перечислении нескольких событий в качестве разделителя используется ключевое слово **OR**.

**ON таблица.** Имя таблицы, модификация которой заданным событием приводит к срабатыванию триггера.

**FOR EACH { ROW | STATEMENT }.** Ключевое слово, следующее за конструкцией FOR EACH и определяющее количество вызовов функции при наступлении указанного события. Ключевое слово ROW означает, что функция вызывается для каждой модифицируемой записи. Если функция должна вызываться всего один раз для всей команды, используется ключевое слово STATEMENT.

**EXECUTE PROCEDURE функция ( аргументы ).** Имя вызываемой функции с аргументами. На практике аргументы при вызове триггерных функций не используются.

### Синтаксис определения триггерной функции

```
CREATE FUNCTION функция () RETURNS trigger AS '
DECLARE
    объявления ;
BEGIN
    команды;
END; '
LANGUAGE plpgsql;
```

В триггерных функциях используются специальные переменные, содержащие информацию о сработавшем триггере. С помощью этих переменных триггерная функция работает с данными таблиц.

Ниже перечислены некоторые специальные переменные, доступные в триггерных функциях

Имя	Тип	Описание
NEW	REC ORD	Новые значения полей записи базы данных, созданной командой INSERT или обновленной командой UPDATE, при срабатывании триггера уровня записи (ROW). Переменная используется для модификации новых записей. <b>Внимание !!!</b> Переменная NEW доступна только при операциях INSERT и UPDATE. Поля записи NEW могут быть изменены триггером.
OLD	REC ORD	Старые значения полей записи базы данных, содержащиеся в записи перед выполнением команды DELETE или UPDATE при срабатывании триггера уровня записи (ROW) <b>Внимание !!!</b> Переменная OLD доступна только при операциях DELETE и UPDATE. Поля записи OLD можно использовать только для чтения, изменять нельзя.
TG_NAME	name	Имя сработавшего триггера.
TG_WHEN	text	Строка BEFORE или AFTER в зависимости от момента срабатывания триггера, указанного в определении (до или после операции).
TG_LEVEL	text	Строка ROW или STATEMENT в зависимости от уровня триггера, указанного в определении.
TG_OP	text	Строка INSERT, UPDATE или DELETE в зависимости от

		операции, вызвавшей срабатывание триггера.
TG_REL ID	oid	Идентификатор объекта таблицы, в которой сработал триггер.
TG_REL NAME	name	name Имя таблицы, в которой сработал триггер.

К отдельным полям записей NEW и OLD в триггерных процедурах обращаются следующим образом: NEW.names , OLD.rg.

### Примеры создания триггеров

**Пример 1.** Триггер выполняется перед удалением записи из таблицы поставщиков s. Триггер проверяет наличие в таблице поставок spj записей, относящихся к удаляемому поставщику, и, если такие записи есть, удаляет их.

-- Создание триггерной функции

```
CREATE FUNCTION trigger_s_before_del () RETURNS trigger AS '
BEGIN
if (select count(*) from spj a where trim(a.ns)=trim(OLD.ns))>0
then delete from spj where trim(spj.ns)=trim(OLD.ns);
end if;
return OLD;
END;
' LANGUAGE plpgsql;
```

-- Создание триггера

```
CREATE TRIGGER tr_s_del_befor
BEFORE DELETE ON s FOR EACH ROW
EXECUTE PROCEDURE trigger_s_before_del();
```

--Проверка работы триггера

```
Delete from s where ns='S2';
```

**Пример 2.** Создание триггера-генератора для таблицы поставщиков s.

Триггер выполняется перед вставкой новой записи в таблицу поставщиков s. Триггер проверяет значения, которые должна содержать новая запись (record NEW) и может их изменить:

- если не указан номер поставщика – он генерируется по схеме – S+ уникальный номер из последовательности;
- если не указано имя поставщика – оно генерируется по схеме – Postawchik\_ + уникальный номер из последовательности;
- если не указан город – ставится значение по умолчанию – “Novosibirsk” ;
- если не указан рейтинг или рейтинг <=0 – устанавливается рейтинг = 10 для поставщиков из Novosibirska и 0 для всех остальных.

-- Создание последовательности

```
CREATE SEQUENCE s_seq INCREMENT BY 1 START WITH 25;
```

-- Создание триггерной функции

-- в этой функции вызывается перегружаемая функция **nlv**, ее определение [здесь](#)

```
CREATE FUNCTION trigger_s_before_ins () RETURNS trigger AS '
BEGIN
NEW.ns=nlv(NEW.ns,'S'||trim(to_char(nextval('s_seq'),'99999')));
NEW.names=nlv(NEW.names,'Postawchik_'||trim(to_char(currval('s_seq'),'99999')));
NEW.town = nlv(NEW.town, 'Novosibirsk' );
if (nlv(NEW.rg,0)<=0) then
```

```

if NEW.town= 'Novosibirsk' then NEW.rg=10;
else NEW.rg=0;
end if;
end if;
return NEW;
END;
' LANGUAGE plpgsql;
    -- Создание триггера
CREATE TRIGGER s_bi
BEFORE INSERT ON s FOR EACH ROW
EXECUTE PROCEDURE trigger_s_before_ins ()

```

```

--Проверка работы триггера
insert into s values(null,null,null,null);
insert into s values(null,null,null,null);
insert into s values(null,null,null,null);
insert into s values(null,'Ivanov',null,null);
insert into s values(null,'Sidorov',50,null);
insert into s values(null,'Petrov',null,'Moskva');

```

--Результат

S25	Postawchik_25	10	Novosibirsk
S26	Postawchik_26	10	Novosibirsk
S27	Postawchik_27	10	Novosibirsk
S28	Petrov	0	Moskva
S29	Ivanov	10	Novosibirsk
S30	Sidorov	50	Novosibirsk