

Севастопольский государственный университет
Кафедра «Информационные системы»

Управление данными

курс лекций

лектор:
ст. преподаватель кафедры ИС Абрамович А.Ю.



Лекция 12

Язык SQL.

Функции и хранимые процедуры.

Генераторы

ФУНКЦИИ И ХРАНИМЫЕ ПРОЦЕДУРЫ В SQL

Функции и хранимые процедуры в SQL, обеспечивают возможность повторного использования и гибкость. Представляют собой блок кода или запросов, хранящихся в базе данных, которые можно использовать снова и снова. Что касается гибкости, в момент, когда происходит изменение логики запросов, можно передавать новый параметр функциям и хранимым процедурам.

Хранимая процедура (ХП) — это программный модуль, который может быть вызван с клиента, из другой процедуры, функции, выполнимого блока (executable block) или триггера. **Хранимые процедуры могут принимать и возвращать множество параметров.**

Функция является программой, хранящейся в области метаданных базы данных и выполняющейся на стороне сервера. К хранимой функции могут обращаться хранимые процедуры, хранимые функции (в том числе и сама к себе), триггеры и клиентские программы. **В отличие от хранимых процедур хранимые функции всегда возвращают одно скалярное значение.** Для возврата значения из хранимой функции **используется оператор RETURN**, который немедленно прекращает выполнение функции.

ФУНКЦИИ

- Функция **имеет возвращаемый тип** и **возвращает значение**
- Использование DML (insert, update, delete) запросов внутри функции невозможно. **В функциях разрешены только SELECT-запросы**
- Функция **не имеет выходных аргументов**
- **Вызов хранимой процедуры из функции невозможно**
- **Вызов функции** внутри **SELECT** запросов **возможен**

ХРАНИМЫЕ ПРОЦЕДУРЫ

- Хранимая процедура **не имеет возвращаемого типа, но имеет выходные аргументы**
- **Использование DML-запросов** (insert, update, delete) **возможно** в хранимой процедуре.
- Хранимая процедура **имеет и входные, и выходные аргументы**
- **Использование или управление транзакциями возможно** в хранимой процедуре
- **Вызов хранимой процедуры** из **SELECT** запросов **невозможно**

Оператор **CREATE FUNCTION** создаёт новую хранимую функцию. Имя хранимой функции должно быть уникальным среди имён всех хранимых функций и внешних функций.

```
CREATE Имя хранимой функции. Может содержать до 63 символов. [or REPLACE] FUNCTION funcname Описание входного параметра. [(Тип данных, который возвращает функция. <inparam>)]  
  RETURNS <type> [COLLATE collation]  
  LANGUAGE plpgsql  
  AS  
  $$  
  DECLARE -- variable declaration  
  BEGIN -- logic  
  END;  
  $$
```

CREATE FUNCTION является составным оператором, состоящий из заголовка и тела. **Заголовок** определяет имя хранимой функции, объявляет входные параметры и тип возвращаемого значения. **Тело функции** состоит из необязательных объявлений локальных переменных, подпрограмм и именованных курсоров, и одного или нескольких операторов, или блоков операторов, заключённых во внешний блок, который начинается с ключевого слова BEGIN, и завершается ключевым словом END.

ВХОДНЫЕ ПАРАМЕТРЫ

```
CREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT DETERMINISTIC LANGUAGE plpgsql
AS
$$
BEGIN RETURN A+B;
END
$$
```

Входные параметры заключаются в скобки после имени функции. Они передаются в функцию **по значению** (любые изменения входных параметров внутри функции никак не повлияет на значения этих параметров в вызывающей программе).

У каждого параметра указывается тип данных (для параметра можно указать ограничение NOT NULL, тем самым запретив передавать в него значение NULL). Для параметра **строкового типа существует возможность задать порядок сортировки** с помощью предложения COLLATE.

Входные параметры **могут иметь значение по умолчанию**. Параметры, для которых заданы значения, должны располагаться в конце списка параметров.

Использование доменов при объявлении параметров

В качестве типа параметра можно указать имя домена. В этом случае **параметр будет наследовать все характеристики домена.**

Если перед названием домена дополнительно используется предложение **TYPE OF**, то используется только тип данных домена — не проверяется (не используется) его ограничение (если оно есть в домене) на NOT NULL, CHECK ограничения и/или значения по умолчанию.

Использование типа столбца при объявлении параметров

Входные и выходные параметры можно объявлять, используя тип данных столбцов существующих таблиц и представлений. Для этого используется предложение **TYPE OF COLUMN**, после которого указывается имя таблицы или представления и через точку имя столбца.

При использовании TYPE OF COLUMN наследуется только тип данных, а в случае строковых типов ещё и набор символов, и порядок сортировки. **Ограничения и значения по умолчанию столбца никогда не используются.**

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

```
CREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT DETERMINISTIC LANGUAGE plpgsql
$$
AS
BEGIN RETURN A+B;
END
$$
```

Предложение RETURNS задаёт тип возвращаемого значения хранимой функции.

Если функция возвращает значение строкового типа, то существует возможность задать порядок сортировки с помощью предложения COLLATE. **В качестве типа выходного значения можно указать имя домена, ссылку на его тип (с помощью предложения TYPE OF) или ссылку на тип столбца таблицы (с помощью предложения TYPE OF COLUMN).**

Детерминированные функции

Необязательное предложение **DETERMINISTIC** указывает, что функция детерминированная.

Детерминированные функции каждый раз возвращают один и тот же результат, если предоставлять им один и тот же набор входных значений. Недетерминированные функции могут возвращать каждый раз разные результаты, даже если предоставлять им один и тот же набор входных значений.

Если для функции указано, что она является **детерминированной**, то такая функция не вычисляется заново, если она уже была вычислена однажды с данным набором входных аргументов, а берет свои значения из кэша метаданных (если они там есть).

В текущей версии Firebird, не существует кэша хранимых функций с маппингом входных аргументов на выходные значения. Указание инструкции **DETERMINISTIC** на самом деле нечто вроде «обещания», что код функции будет возвращать одно и то же. В данный момент детерминистическая функция считается инвариантом и работает по тем же принципам, что и другие инварианты (вычисляется и кэшируется на уровне текущего выполнения данного запроса).

ТЕЛО ФУНКЦИИ

```
CREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT DETERMINISTIC LANGUAGE plpgsql
AS
$$
BEGIN RETURN A+B;END
$$
```

После ключевого слова **AS** следует тело хранимой функции.

В необязательной секции **<declarations>** описаны локальные переменные функции, именованные курсоры и подпрограммы (подпроцедуры и подфункции). Локальные переменные подчиняются тем же правилам, что и входные параметры функции в отношении спецификации типа данных.

После необязательной секции деклараций обязательно следует **составной оператор**. Составной оператор состоит из одного или нескольких операторов, заключенных между **ключевыми словами BEGIN и END**. Составной оператор может содержать один или несколько других составных операторов. **Вложенность ограничена 512 уровнями**. Любой из **BEGIN ... END** блоков может быть пустым, в том числе и главный блок

Кто может создать функцию?

Выполнить оператор CREATE FUNCTION могут:

- администраторы;
- пользователи с привилегией CREATE FUNCTION.

Пользователь, создавший функцию, **становится её владельцем**.

ПРИМЕР: запросить имя пользователя и его самые дорогие покупки.

```
sbrmvch=# select * from purchases;
```

id	name	cost	user_id
1	M1 MacBook Air	1300.99	1
2	Iphone 14	1200.00	2
3	Iphon 10	700.00	3
4	Iphone 13	800.00	1
5	Intel Core i5	500.00	4
6	M1 MacBook Pro	1500.00	5
7	IMAC	2500.00	7
8	ASUS VIVOBOOK	899.99	6
9	Lenovo	1232.99	1
10	Galaxy S21	999.99	2
11	XIAMI REDMIBOOK 14	742.99	4
12	M1 MacBook Air	1299.99	8
13	ACER	799.99	7

(13 rows)

```
sbrmvch=# select * from users;
```

id	name	profession
1	Bob	QA
2	Camilo	Front End developer
3	Billy	Backend Developer
4	Alice	Mobile Developer
5	Kate	QA
6	Wayne	DevOps
7	Tim	Mobile Developer
8	Amigos	QA

(8 rows)

```
CREATE OR REPLACE FUNCTION findMostExpensivePurchase(customer_id int)
    RETURNS numeric(10, 2)
    LANGUAGE plpgsql
AS
$$
DECLARE
    itemCost numeric(10, 2); — объявляем локальную переменную;
begin
    SELECT MAX(cost) INTO itemCost — инициализируем переменную;
    FROM purchases
    WHERE user_id = customer_id;
    RETURN itemCost; — возвращает значение функции;
end;
$;
```

Чтобы вызвать функцию — необходимо выполнить следующую команду:

```
SELECT findMostExpensivePurchase(1) as mostExpensivePurchase;
```

```
[sbrmvch=# SELECT findMostExpensivePurchase(1) as mostExpensivePurchase;
```

```
mostexpensivepurchase
```

```
-----
```

```
1300.99
```

```
(1 row)
```

```
SELECT name, findMostExpensivePurchase(id) as purchase
       from users;
```

```
[sbrmvch=# SELECT name, findMostExpensivePurchase(id) as purchase from users;
```

```
name | purchase
```

```
-----+-----
```

```
Bob   | 1300.99
```

```
Camilo | 1200.00
```

```
Billy | 700.00
```

```
Alice | 742.99
```

```
Kate  | 1500.00
```

```
Wayne | 899.99
```

```
Tim   | 2500.00
```

```
Amigos | 1299.99
```

```
(8 rows)
```

ALTER FUNCTION

Оператор **ALTER FUNCTION** позволяет **изменять состав и характеристики входных параметров**, типа выходного значения, локальных переменных, именованных курсоров, подпрограмм и тело хранимой функции.

```
ALTER FUNCTION funcname [(<inparam> [, <inparam> ...])]
```

```
RETURNS <type> [COLLATE collation]
```

```
[DETERMINISTIC]
```

```
<routine-body>
```

Будьте осторожны при изменении количества и типов входных параметров хранимых функций. Существующий код приложения может стать неработоспособным из-за того, что формат вызова функции несовместим с новым описанием параметров.

Выполнить оператор ALTER FUNCTION могут:

- администраторы;
- владелец хранимой функции;
- пользователи с привилегией ALTER ANY FUNCTION.

DROP FUNCTION

```
DROP FUNCTION funcname
```

Оператор **DROP FUNCTION** **удаляет существующую хранимую функцию**. Если от хранимой функции существуют зависимости, то при попытке удаления такой функции будет выдана соответствующая ошибка.

Оператор **CREATE PROCEDURE** создаёт новую **хранимую процедуру**. Имя хранимой процедуры должно быть уникальным среди имён всех хранимых процедур, таблиц и представлений базы данных.

```
CREATE [OR REPLACE] PROCEDURE procedure_name(parameter_list)
LANGUAGE language_name
AS $
    stored_procedure_body;
$;
```

Создание хранимой процедуры, почти такое же, как создание функции с небольшим отличием — **в ней нет return**. Остальное почти идентично.

CREATE PROCEDURE является составным оператором, состоящий из **заголовка и тела**. Заголовок определяет имя хранимой процедуры и объявляет входные параметры.

Тело процедуры состоит из **необязательных объявлений** локальных переменных, подпрограмм и именованных курсоров, и одного или нескольких операторов, или блоков операторов, заключённых во внешний блок, который начинается с ключевого слова **BEGIN**, и завершается ключевым словом **END**.

Функции позволяют выполнять только SELECT-запросы, а хранимые процедуры позволяют выполнять INSERT, UPDATE, DELETE операции. **Хранимые процедуры очень удобны при работе со случаями, когда необходимы операции INSERT, UPDATE ИЛИ DELETE.**

ПРИМЕР: банковские переводы.

```
[sbrmvch=# select * from accounts;
```

id	balance	user_id
1	1500	1
2	1100	2
3	2300	3
4	7500	5
5	6500	4

(5 rows)

```
[sbrmvch=# select * from users;
```

id	name	profession
1	Bob	QA
2	Camilo	Front End developer
3	Billy	Backend Developer
4	Alice	Mobile Developer
5	Kate	QA
6	Wayne	DevOps
7	Tim	Mobile Developer
8	Amigos	QA

(8 rows)


```
CREATE OR REPLACE PROCEDURE transfer(sourceAccountId bigInt,  
destinationAccountId bigInt, amount Integer)  
language plpgsql  
as $$  
begin  
    update accounts  
    set balance = accounts.balance - amount  
    where id = sourceAccountId;  
  
    update accounts  
    set balance = balance + amount  
    where id = destinationAccountId;  
  
commit;  
end;  
$$;
```

В приведенном примере показано **создание процедуры** — **transfer()**, которая принимает три параметра. Сразу после имени процедуры передаются аргументы с соответствующими типами данных — **sourceAccountId**, **destinationAccountId**, **сумма**. **Процедура вычитает переданную сумму из одного account и добавляет ее к другому account.**

Для вызова хранимой процедуры используется — `call procedure_name()` .

```
call transfer (5, 4, 2000);
```

```
[sbrmvch=# call transfer (5, 4, 2000);
```

```
[sbrmvch=# select * from accounts;
```

id	balance	user_id
1	1500	1
2	1100	2
3	2300	3
4	7500	5
5	6500	4

(5 rows)



```
[sbrmvch=# select * from accounts;
```

id	balance	user_id
1	1500	1
2	1100	2
3	2300	3
5	4500	4
4	9500	5

(5 rows)

ГЕНЕРАТОРЫ (ПОСЛЕДОВАТЕЛЬНОСТЬ)

объект базы данных, предназначенный для получения уникального числового значения.

Генераторы используют для создания автоинкрементных полей. Для каждого такого поля придется создавать собственный генератор. *Генератор гарантирует, что значение этого поля всегда будет уникальным.*

```
CREATE {SEQUENCE | GENERATOR} <name_GEN>;
```

Операторы **CREATE SEQUENCE** и **CREATE GENERATOR** являются синонимами – *оба создают новую последовательность*. Можно использовать любое из них, но рекомендуется использовать CREATE SEQUENCE, если важно соответствие стандарту.

В момент создания генератора ему устанавливается значение равное 0.

Значение генератора изменяется также при обращении к функции **GEN_ID**, где в качестве параметра указывается имя последовательности и значение приращения.

```
GEN_ID(name_GEN, step);
```

name_GEN – имя генератора; *step* – шаг, на который требуется увеличить значение.

ГЕНЕРАТОРЫ (ПОСЛЕДОВАТЕЛЬНОСТЬ)

Следует быть крайне аккуратным при таких манипуляциях в базе данных, они могут привести к потере целостности данных. Если `step` равен 0, функция не будет ничего делать со значением генератора и вернет его текущее значение.

Оператор **NEXT VALUE FOR** возвращает следующее значение в последовательности.

NEXT VALUE FOR <name_GEN>

NEXT VALUE FOR не поддерживает значение приращения, отличное от 1.
Если требуется другое значение шага, то используется старая функция GEN_ID.

Оператор **ALTER SEQUENCE** устанавливает значение последовательности или генератора в заданное значение.

ALTER SEQUENCE <name_GEN> RESTART WITH <new_val>

Оператор **SET GENERATOR** устанавливает значение последовательности или генератора в заданное значение.

SET GENERATOR <name_GEN> TO <new_val>

*оператор считается устаревшим и оставлен ради обратной совместимости

Для просмотра текущего значения генератора, необходимо выполнить команду:

SELECT GEN_ID(<name_GEN>, 0) FROM RDB\$DATABASE;

ГЕНЕРАТОРЫ (ПОСЛЕДОВАТЕЛЬНОСТЬ)

RDB\$DATABASE – это системная таблица, которая хранит основные данные о базе данных, она всегда существует во всех базах данных Firebird и всегда содержит только одну строку.

RDB\$GENERATORS – это системная таблица, которая хранит сведения о генераторах (последовательностях).

Основное простое правило по переустановке значений генераторов в работающей базе данных –
не делать этого.

Удаление последовательности (генератора):

DROP {SEQUENCE | GENERATOR} <name_GEN>

Операторы ***DROP SEQUENCE*** и ***DROP GENERATOR*** эквивалентны: оба удаляют существующую последовательность (генератор).

Удалить генератор может либо его владелец либо SYSDBA при условии что его нет в зависимостях других объектов.